

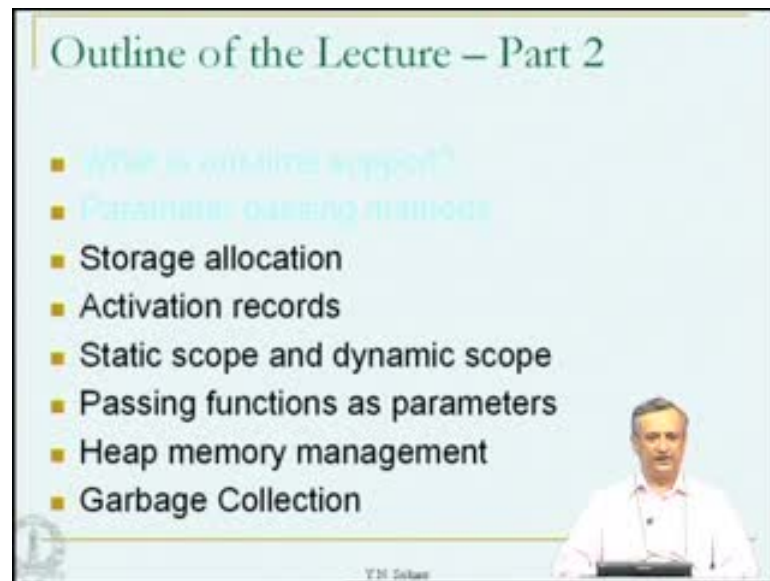
Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 02

Lecture No. # 03

Run-time Environments-Part 2

(Refer Slide Time: 00:25)



The slide is titled "Outline of the Lecture – Part 2" and contains a list of eight topics. The first two topics, "What is runtime support?" and "Parameter passing mechanisms", are highlighted in light blue. The remaining six topics are in black text. A small inset image of the professor is visible in the bottom right corner of the slide.

- What is runtime support?
- Parameter passing mechanisms
- Storage allocation
- Activation records
- Static scope and dynamic scope
- Passing functions as parameters
- Heap memory management
- Garbage Collection

Welcome to part 2 of this lecture on run-time environments. In the last lecture, we discussed some aspects of run-time support; for example, what is run-time support, parameter passing mechanisms and a little bit of storage location.

(Refer Slide Time: 00:45)

Static Data Storage Allocation

- 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

The diagram on the right shows a vertical stack of four green boxes representing memory allocation for variables: 'Main program variables', 'Procedure P1 variables', 'Procedure P2 variables', and 'Procedure P4 variables'. A small inset image of a man in a white shirt is visible in the bottom right corner of the slide.

Today, we will continue the part 2 of the lecture with the topics that we have not yet discussed. A little bit of review from last time: we discussed static data storage allocation. Basically, the compiler is responsible for allocating space for all variables both local and global in this type of allocation.

This is done for all procedures at compile time. If you look at the picture here, you have main program variables, procedure P 1 variables, P 2, P 3 variables and so on and so forth.

The difficulty with this allocation method is we cannot have recursion. No stack and heap allocation is available here. There are no overheads, but at the same time we cannot have recursion.

(Refer Slide Time: 01:35)

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

Y.H. Sakar

Dynamic storage allocation is another policy, where compiler locates place only for global variables at compile time, but space for all other variables of procedures will be allocated only at runtime. This policy is based on stack heap allocation and it is available in C, C plus plus, Java, FORTRAN 8 and FORTRAN 9.

The variable access in this case is a bit slow because there are several indirections in accessing locations, but recursion can be implemented here. This is the biggest advantage in dynamic storage location.

(Refer Slide Time: 02:17)

Activation Record Structure

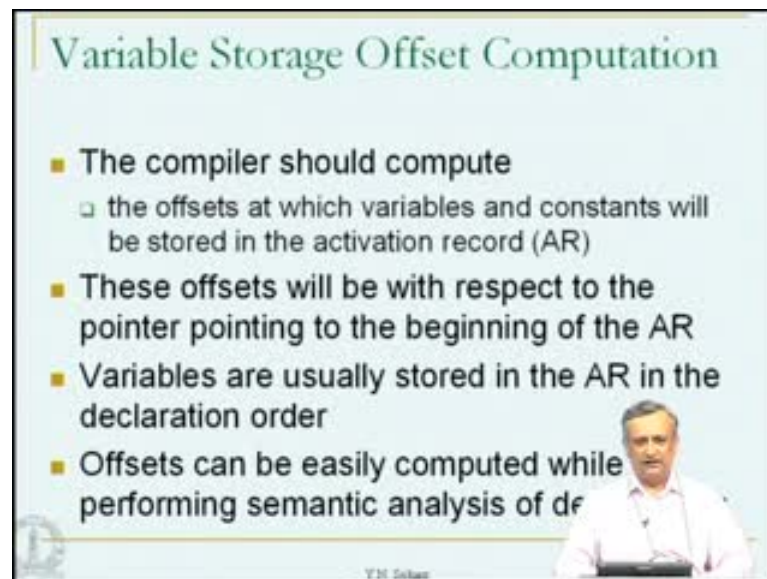
Return address
Static and Dynamic links (also called Access and Control link resp.)
(Address of) function result
Actual parameters
Local variables
Temporaries
Saved machine status
Space for local arrays

Note:
The position of the fields of the act. record as shown are only notional.
Implementations can choose different orders; e.g., function result could be at the top of the act. record

Y.H. Sakar

We also saw the activation record structure. An activation record is a unit of allocation for the run-time support. Whenever a procedure is called, a new activation record is created; parts of the activation record are initialized by the caller and parts of the activation record are initialized by the callee. There is almost everything that you need, all the information: return address, some static and dynamic links, address of the function result and then the actual parameters, local variables, temporaries, machine status, that is, the set of registers and other information, space for local arrays, etcetera.

(Refer Slide Time: 03:06)



The slide, titled "Variable Storage Offset Computation", lists the following points:

- 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 de

The slide also features a small inset image of a man in a white shirt and a logo in the bottom left corner.

The first thing that a compiler should do during intermediate code generation itself is to compute the offset at which variable is going to be stored in the activation record. If the storage allocation is static, then these are going to be offsets in the static data area itself, but otherwise these are all offsets in the activation record. The compiler should compute the offsets at which variables and constant will be stored in the activation record.

These offsets will be with respect to the pointer pointing to the beginning of the activation record and variables are usually stored in the activation record in the declaration order itself. For example, if you have `int A, B, C;` `float D, F;` something like that, then the offsets for A, B, C will be 2, 0, 4 and 8; and the next D and F will get offset from 8 onwards that is 12 and 12 plus 8; that is 20. It is in the storage in on declaration order itself. We will see a little more detail of how is this computation is performed during semantic analysis.

(Refer Slide Time: 04:35)

Example of Offset Computation

```
P → Decl { Decl.inoffset↓ = 0; }
Decl → T id ; Decl1
      { enter(id.name↑, T.type↑, Decl.inoffset↓);
        Decl1.inoffset↓ = Decl.inoffset↓ + T.size↑;
        Decl.outoffset↑ = Decl1.outoffset↑; }
Decl → T id ; { enter(id.name↑, T.type↑, Decl.inoffset↓);
                Decl.outoffset↑ = T.size↑; }
T → int { T.type↑ = inttype; T.size↑ = 4; }
T → float { T.type↑ = floatype; T.size↑ = 8; }
T → [num] T1 { T.type↑ = arraytype(T1.type↑, T1.size↑);
                T.size↑ = T1.size↑ * num.v; }
```

YH Sakar

For example, consider a simple grammar program consisting of only declarations. P going to decl and a declaration is type T followed by id semicolon Decl 1; Decl 1 is the other non-terminal which produces more and more declarations. Otherwise the ending terminal production is decl going to T type and followed by id.

Type can be int, float or num as indicated in the other 3 productions here. Let us start looking at the method by which the variable offset is computed. The declaration is the most important non terminal. It has actually an inherited attribute called inoffset and it also has a synthesise attribute called outoffset.

To begin with, at the beginning of the declaration list this inoffset is set to 0 and then as we parse declarations, for example, look at this now (Refer Slide Time: 05:52) T id semicolon declaration 1. This identifier is entered into the symbol table along with the name, the type and the inoffset that is the beginning of that declaration list there is some offset which comes in to this list. That is the offset which is assigned to the name id. Now the declaration 1 dot inoffset is also set as declaration dot inoffset, but it is added an increment of T dot size.

The reason is this declaration id is of type T and T dot size is the size of that particular type. For example, ints may take 4 bytes; floats may take 8 bytes and so on and so forth. That is why, whatever we got from the left hand side declaration non terminal is

incremented by the size of that particular type and that is assigned to declaration 1 dot inoffset.

What is declaration dot outoffset? That is whatever we get from declaration 1 dot outoffset. The declaration which are parsed by this non terminal or generated by this non terminal are all accumulated and then outoffset will give you the final offset for this particular thing.

This recursive production applies itself again and again and generates all the declarations. Each time a declaration is produced there is some incrementing and then the next declaration gets that added offset. For example, declaration going to T followed by id terminates this recursion and at that point we enter the name with type and some offset and outoffset is T dot size itself because this is a declaration which is terminating the entire thing.

This is how we parse the declarations and assign variable offsets.(Refer Slide Time: 08:00) T going to int - size is 4, T going to float - size is 8 and T going to num T 1 is an array. The T dot type is array type and T dot size is T 1 dot size into the num dot value that is the size of that particular array is num dot val. This num dot value is multiplied by the type of this particular array whose size is available in T₁ dot size. This is an example of how variable offset computation is performed in the declaration order.

(Refer Slide Time: 08:36)

Allocation of Activation Records

```
program RTST;  
procedure P;  
  procedure Q;  
    begin R; end  
  procedure R;  
    begin Q; end  
  begin R; end  
begin P; end
```

SL chain Static Link
Base Dynamic Link
Next RTST DL chain

Activation records are created at procedure entry time and destroyed at procedure exit time

RTST -> P -> R -> Q -> R

Y.H. Sakar

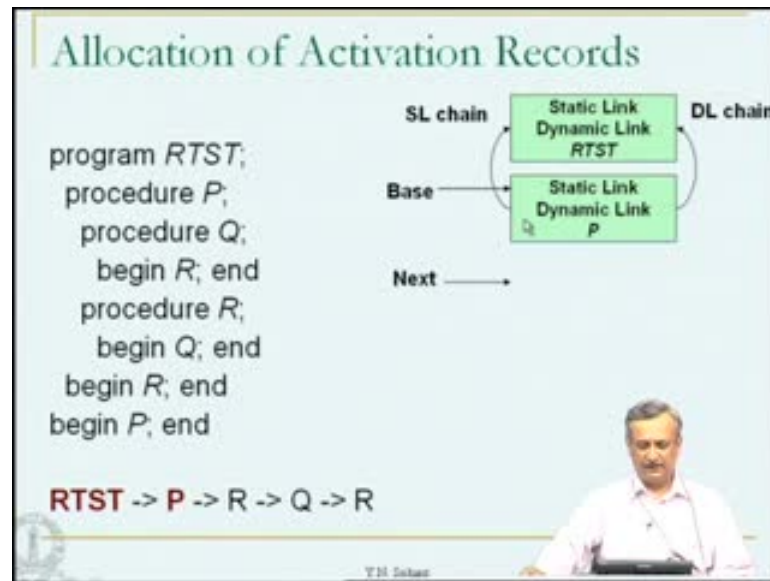
Now we come up to a very important part of this lecture, how are activation records allocated during dynamic allocation that happens in languages such as C and C plus plus. Let us consider a simple program here, program RTST and the procedure is P, which is actually enclosed within the main program RTST. Then procedure Q is enclosed within procedure P and the procedure Q has a body in which we say begin, call the procedure R and end; procedure R is just outside procedure Q; it also has a body which says begin, call the procedure Q and end.

So, basically Q calls R and R calls Q there is some recursion which happens here and at the level of procedure P itself, there is a body which says call R and the main program begins by calling P. Here is a call sequence which is possible. The main program RTST calls P and P calls R, R calls Q and Q calls R.

We are going to trace the allocation of activation records for this particular sequence. What we need to remember at this point is activation records are created at procedure entry time and they are destroyed at procedure exit time. (Refer Slide Time: 10:12) To begin with, what we have is the main program space in the stack of activation records. In the stack heap area we just have the main program data already available because these are like global variables. Whatever is stored here is a set of global variables which can be accessed by any of these procedures; that is the program variables which are declared here.

There is a static link chain which will be shown in the following diagram on this side and there is a dynamic link chain which will be shown in the diagram on the other side. So, let see. Next is the next place where the activation record can be created in the stack heap.

(Refer Slide Time: 10:58)



So, RTST has called P, the activation record for P did not exist; it is created now. Now there are 2 questions to be answered: One is what to do with the dynamic link chain and the other one is what to do with the static link chain. The rest of the initialization within the activation record is fairly straight forward and there is nothing much involved there; various return addresses and temporaries etcetera are all stored there.

The dynamic link chain is used only to maintain the stack nature of this particular set of activation records. As we go on, as we add 1 activation record at a time, this dynamic link will point to the previous activation record. When we remove for example, this particular activation record, the activation for P ends after the call chain comes back. This entire thing will be removed; this particular link also disappears.

What is the static link? The static link actually allows us to access the global variables. For example, if you look at the scope of the variables which are visible within the procedure P, here is the body of the procedure P, this particular body can access the variables of program RTST; they can also access the variables of the procedure P.

As far as the variables of program RTST are concerned, they are like global variables and the variables within the procedure P are the local variables. The access to local variables is very easily done with respect to a pointer call the base pointer. All the offsets, we saw offset computation example before, offsets are all with respect to this

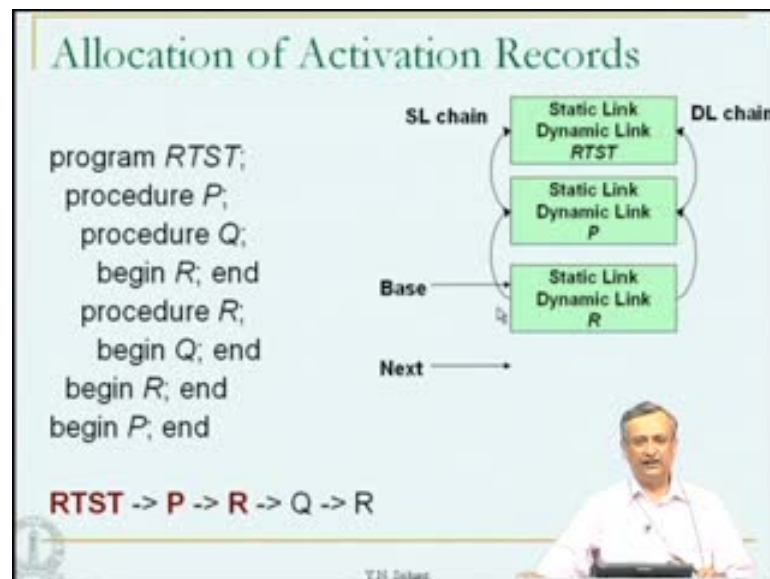
base pointer. So, it is very easy to add the offset to the base pointer and get to the location where the local data is stored.

For each variable, we have an offset and adding that to this base will give us the location where the variable is stored. Suppose we have global variables that is the variables of the main program RTST which we need to access within the procedure P, in such a case, the static link chain is going to help us. Take the static link, see where it points, it is actually pointing to RTST itself.

If we take the contents of this particular static link variable, we get the base pointer of this particular RTST (Refer Slide Time: 13:45); that is the way it would be. Once we know the base pointer of RTST, accessing the variable within that is really straight forward. Again we have the offset that has already been computed and it is easy to access it.

All this happens at compile time. In other words, the compiler already knows because of the nesting levels of the procedures, how many times it needs to skip the static link. This will become clear as we go along and then treat that as a pointer to the activation record and access the variables within that activation record.

(Refer Slide Time: 14:28)



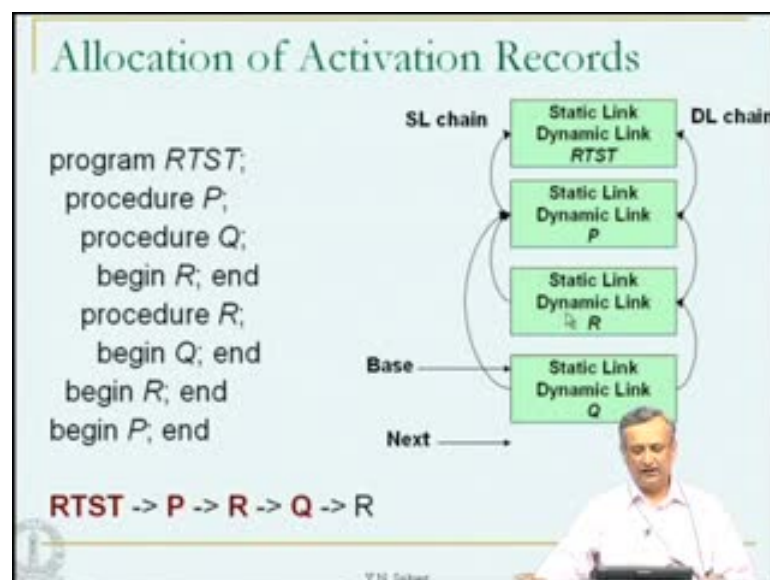
Let us now look at the call to R from P. See the activation record for C is now created fresh. The dynamic link simply points to the previous one, previous activation record that is the tough P and then whatever remains as dynamic link from P to RTST remains.

This is the chain of dynamic links used to maintain the stack nature of the activation records. On this side look at the procedure R; the procedure R is just within the procedure P. So, the body of the procedure R can access the variables which are within the procedure R, it can access the variables of the procedure P which is the enclosing procedure and it can also access the variables of RTST which is the next level enclosing procedure - that is the main program itself.

Using the base pointer, the code can access all the variables of R by using one level of indirection on the static link and then treating that as the base pointer it can access all the variables of P. By doing indirection twice, it can get the base pointer of the main program RTST and access the variables of RTST.

Remember that the compiler itself knows how many times this indirection needs to be applied because the symbol table will contain the nesting depth of each one of the program variables and the procedures.

(Refer Slide Time: 16:12)

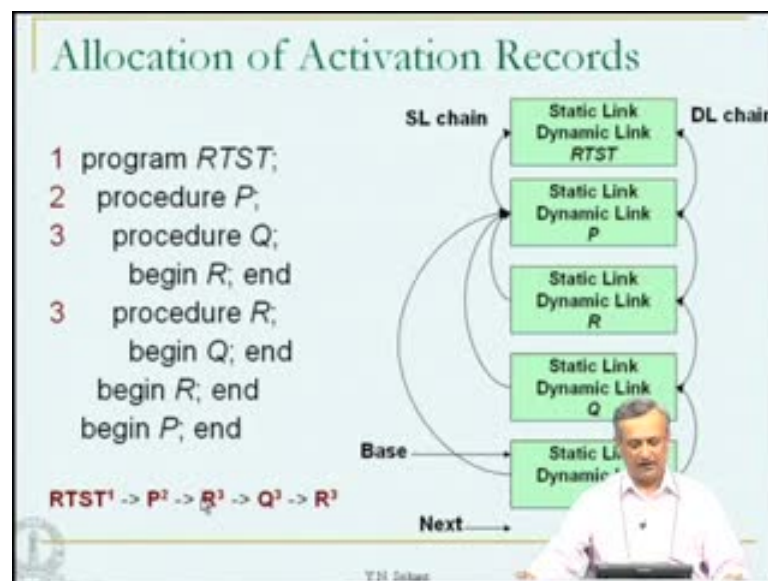


We go one step further, R calls Q. We had RTST which called P; then P called R; now R calls Q. R and Q happen to be at the same nesting level. They are not nested within one another.

If you look at Q, the body of Q can access the variables within procedure Q, it can access variables within the procedure P and it can access the variables within the main program RTST. That is all, just like the procedure R. It is not allowed to access the variables of procedure R. So, making the static link point blindly to the activation record of R will be incorrect. If that is done, then the variables of R can also be accessed which is not right.

Therefore, this particular static link (Refer Slide Time: 17:05) should be made to point to the beginning of the activation record of P itself. Now, whenever the compiler needs, it can generate code to access the variables of this procedure Q. By doing one indirection, it can access the variables of P and by doing 2 indirections, it can access the variables of RTST. The variables of R will not be accessed at all, which is the right thing to do.

(Refer Slide Time: 17:34)

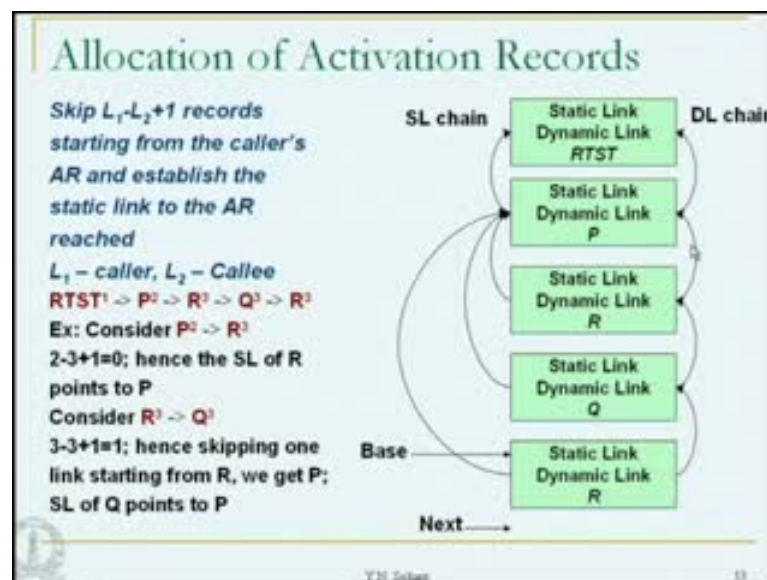


Let us look at the next activation which is that of R. So far, we had RTST then called P, P calls R, R calls Q and then there is a recursion, Q calls R again. There are 2 instances of R and an instance of Q. So, R cannot and should not access the variables of Q - this R, the second instance of R and it should not also access the variables of the first instance of R. It should actually be accessing only the variables of R, P and RTST and this is appropriately done by making the static pointer or the static link of this particular

activation record point to the beginning of the activation record of P. The next thing that we need to see is how exactly are we going to establish these static links. Dynamic links are very easy. This is dependent on the level information of the various procedures.

For example, RTST is assumed to be at level 1; procedure P is assumed be at level 2; then procedure Q is at level 3; procedure R is also at level 3. The levels of the procedures are all indicated in this caller chain. RTST has superscript 1, P has 2, R has 3, Q has 3 and R has 3.

(Refer Slide Time: 19:02)



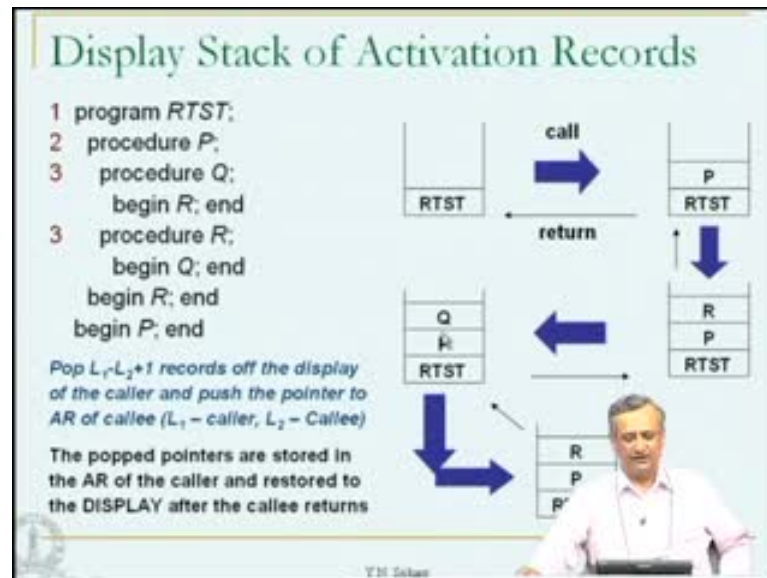
Let us see how it is established. The basic rule is very simple: skip L_1 minus L_2 plus 1 records starting from the caller's activation record and establish the static link to the activation record reached.

What is L_1 ? L_1 is the level of the caller and L_2 is the level of the callee. Let us take this call chain and look at an example, 2 examples really. Consider the call P to R; that means this RTST would already be present; P will already be present, but R is not yet present; of course, these (Refer Slide Time: 19:43) 2 are not yet present.

So, the level difference is 2 minus 3 plus 1 which is 0. Hence, the static link of R should point to this activation record of P itself. That is the simplest rule. Similarly, take R calls Q; we should really be pointing here. Let us see whether we do that.

So, 3 minus 3 plus 1 is 1. Hence, skipping one link starting from R, we get P and the static link of Q will point to the activation record of P. This is how the static link chain is set and this formula, which is very simple one, enables us to set this links appropriately.

(Refer Slide Time: 20:34)



What we saw so far is one method of arranging activation records; this is using the static link and dynamic link. There is another way of establishing access to these global variables and arranging the activation records; that is known as the display stack.

Let us discuss this display stack and then see what is the advantage or otherwise of each of these schemes. Take the same program RTST, P, Q and R. The display stack is a very simple stack of pointers to the activation records. To begin with the display stack has a pointer to the activation record of RTST, the main program and nothing else. Activation records are going to be stored exactly the way they were stored before - that is in the stack heap area.

The creation of activation record, destruction of activation record etcetera happens as before; there is no difference at all. To begin with, we have just this RTST; one pointer pointing to the activation record of RTST.

Now there is a call; the call is to the procedure P. When there is a call, we actually push the activation record pointer of the callee or we decide to pop some of the entries of the

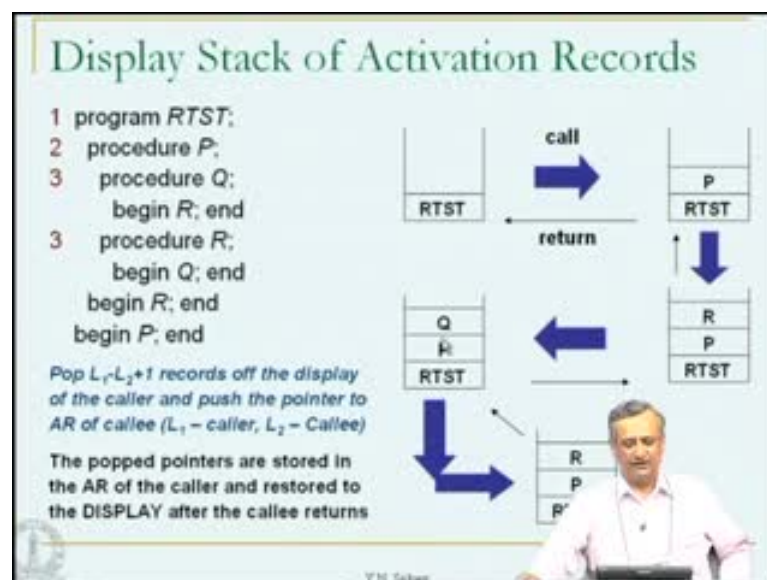
display stack and then push the activation record pointer; one of these can happen based on this particular formula.

The formula says: Pop L_1 minus L_2 plus 1 records off the display of the caller and push the pointer to the activation record of callee on to the stack. L_1 is the caller, L_2 is the callee and the formula is exactly the same as before.

Let us do that here. The levels, when we look at it, RTST is at level 1, P is at level 2; 1 minus 2 plus 1 is 0. So, nothing needs to be popped from this display stack; you only push the pointer to P's activation record. Then P calls R; P is at level 2 and R is at level 3. 2 minus 3 is minus 1 and then plus 1, again 0. So, we again need not pop anything from this display. You only push the activation record pointer of R on to this display stack, but now something different happens. R now calls Q, both are at level 3. 3 minus 3 is 0 plus 1 is 1.

So, we need to pop one activation record pointer; that is that of R from this stack which uncovers the pointer P and then we push the new pointer Q on to the stack exactly the way we did before. There is no difference at all. There, the static link changes allowed us to access the various global variables.

(Refer Slide Time: 20:34)



Here, as I am going to tell you very soon, the stack of activation record pointers - the display stack allows us to do exactly the same thing. We get 3 activation record pointers

Q, P and RTST on this and then once this happens, there is another call to R again; 3 minus 3 plus 1 is 1. So, one of the pointers that is Q is pushed out R. The new R activation record pointer is pushed on to the stack and from now onwards the recursion unwinds. This is where this particular comment becomes important.

The popped pointers are stored in the activation record of the caller and restored to the display after the callee returns. In other words, whatever we popped here cannot be just dispensed with. We need to store it in the activation record of the caller and then once we return, this particular pointer will have to be pushed on to the stack again and situation should be exactly as before.

This is after the callee returns; that is what we do. The callee will not know about this activation record pointers at all. See for example, when R calls Q we have dispensed with the activation record of R, rather the pointer to activation record of R is removed from here and then the pointer to the AR of Q is inserted. Therefore, the callee cannot store this pointer in its memory, only the caller will have to store the pointers which are going to be popped off and then place the call to the new procedure.

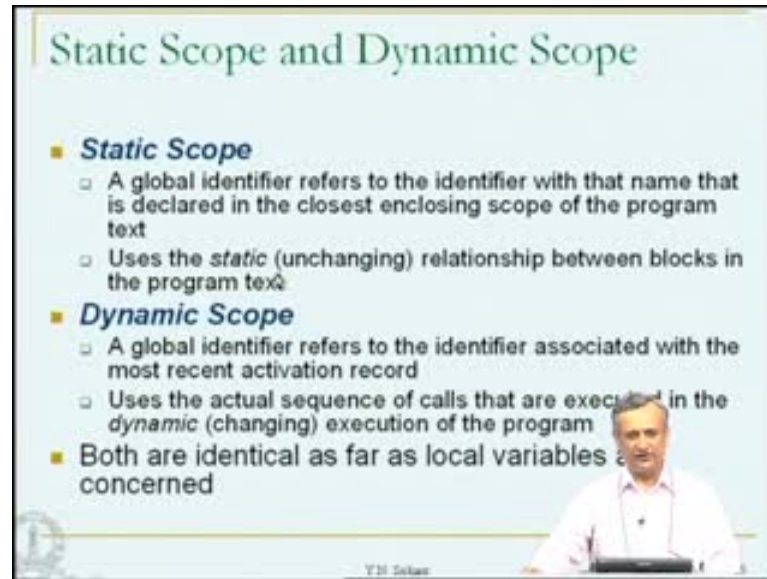
Let us see how the variables are accessed within these programs. Let us take this example. We have Q, we have P, we have RTST. These are the 3 activation records which are active at this point of time and the code of procedure Q can access the variables of itself, those of P and those of RTST; that is what this really means.

This can be done very easily here. Q is a pointer to the activation record; this is the pointer to the Qs activation record. Using this pointer, it can access the AR of Q and variables of Q with an offset.

Similarly at the next level, if there is a variable of P which is easily known from the symbol table, the compiler generates code to take the pointer AR which is mentioned here as P, go to that particular activation record, use the offset and dig into the AR of P. Similarly for RTST also.

The access is very similar to what happened before. Only thing is we really do not have to traverse any static link chains. Here, we just use these pointers which are on the display stack in order to access the variables.

(Refer Slide Time: 27:04)



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

YH Sakar

Before we begin this static scope and dynamic scope, let me mention a few points about the display stack and the static link, dynamic link chain itself - the methods themselves. When you have the sldl scheme, you need to traverse the static link chains possibly many times before we access the right global variables.

There are many indirections which can happen. This can become a bit slow if the number of global variables in the program is large. This does not happen in the case of the display stack. The pointers are already on the stack and taking them from the stack is no overhead, but popping and pushing the pointers from the display stack is extra in the case of the display stack mechanism. This does not happen in the case of the sldl scheme.

It is very difficult to say which particular method is more efficient compared to the other and there are compilers which use either of these methods. We are not going to get in to a debate on which is better; both are equally good and any one can be used

Let us proceed further; the next concept that we discuss now relates to the scope. There are 2 varieties of scope: first is the static scope and the second is the dynamic scope. Let us understand these words well. When the scope is said to be static, a global identifier or a name, global name refers to that particular identifier with that name that is declared in the closest enclosing scope of the program text. It uses the static relationship between blocks in the program text.

That is, this is exactly the way we understand the scope of variables in C and C plus plus. Look at the program, see at which level the variable is declared and whenever we want to refer to a variable, we just go up the nesting levels, find the first place where it is declared and that is our variable.

That is, what we mean by saying the name refers to the identifier with that name that is declared in the closest enclosing scope of the program text. You just go up the nesting levels of for the procedures, find the first one which contains that particular name. This particular relationship does not change. It is related to the program text and it is not related to the execution of the program but dynamic scope is different.

We are going to see example of this very soon. A global identifier refers to the identifier associated with the most recent activation record. For example, if we have 2 procedures with the same variable x depending on which is active a point, x may refer to either one of them. We will see an example of this. It uses the actual sequence of calls that are executed in the dynamic execution of the program and both are identical as far as local variables are concerned; there is no different at all.

(Refer Slide Time: 30:40)

**Static Scope and Dynamic Scope :
An Example**

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

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

Stack of activation record-
after the call to g

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

The slide illustrates the state of the program after a call to function g. It shows three activation records in a stack. The outer block has x=1. The call to f(3) has y=3 and x=4. The call to g(12) has z=12. The dynamic scope of x is 4, which is the value in the most recent activation record (f(3)).

Let us see a simple example. Here is a C like program: int x equal to 1; function g z, the body is x plus z, it is just an expression; function f y has a bigger body, it has a variable x equal and it is assigned the value y plus one and it returns g of y star x; this particular function f is called with f 3 in the main program.

Let us assume static scope. What really happens is, when we are within this function g z the x here, there is no declaration of x within the function g ; it just goes to the next nesting level that is, the main program. This particular x is what we refer here; that is static scope.

When we come to a function f y , you have $\text{int } x$ equal to y plus 1; this is a local variable which is declared within the function f . Then we say $\text{return } g$ of y star x . We have called f with 3. So, y is 3, x becomes 4 and g is called with 4 into 3 - that is 12.

When we come to g z , let us trace what happens. This says x plus z , z is already 12, but the x we refer to is 1. So, 12 plus 1, 13 is returned as the value of f . This is the way a program executes when it is using static scope.

Most importantly, even though the function f y is active, when g is executing; for example, we called f . See, you look at this particular chain (Refer Slide Time: 32:43): This is the outer block x with value one; this is the activation record corresponding to f of 3; the variable y and variable x are present here; y is the parameter, x is the local variable; y has the value 3, x has the value 4 and then we have another activation record corresponding to g where the value of z , which is the parameter, is 12.

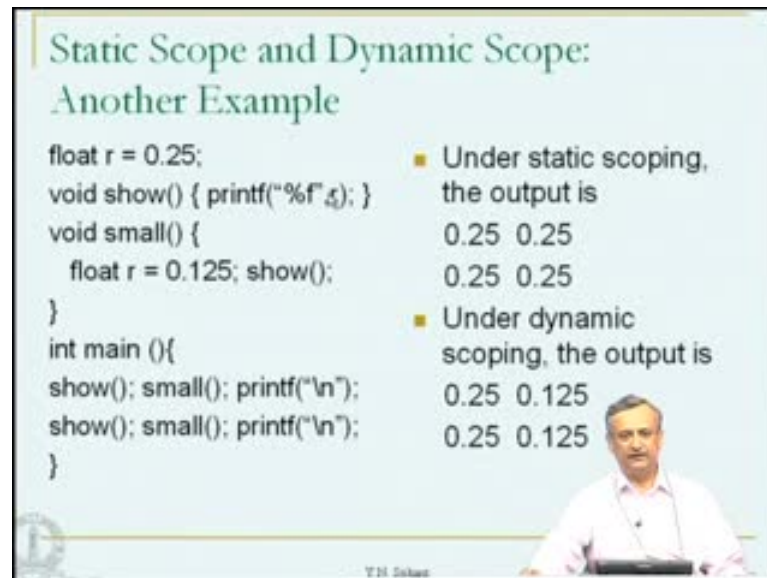
Let us see what happens if we use dynamic scope. Let us trace the whole thing all over again we called f with 3; x becomes 3 plus 1 4; then g is called with y star x that is g of 12. So far so good, there is no change with what happened.

But when the function g is called it says x plus z . In dynamic scope, the question asked is which x ? In static scope, it was very easily resolved as this particular outer x , the enclosing main program is looked at and x is determined to be the variable which is relevant here. In dynamic scope, it looks at the most recent activation of some function which contains the variable declaration for x . In this case, it is this particular f . So, that is why it is so. Let me go back.

We have g here, the parameter is 12, but when we come here x plus z , the most recent activation of x corresponds to the variable $\text{int } x$ within the function f y here. (Refer Slide Time: 34:32) This x is not this x which is relevant, when dynamic scope is active. In static scope, this is always the variable that is relevant whereas, in dynamic scope this is the most recent activation of x ; so this x is relevant.

What we do is add 4 to 12 and return 16. In the previous case, we had added one to 12 and returned 13. Now, we return 16. This is the precisely the way the dynamic scope works.

(Refer Slide Time: 34:55)



The slide contains the following text:

Static Scope and Dynamic Scope: Another Example

```
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");
}
```

- 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

The slide also features a small inset image of a man in a white shirt speaking at a podium.

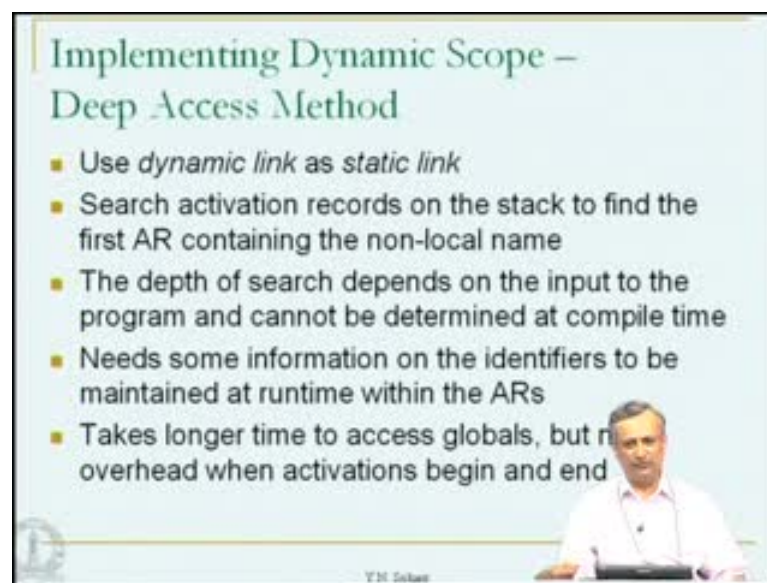
Here is another example. Let us see what happens. Here is the floating variable r , 0.25; then the procedure `show` simply prints the value of r ; the procedure `small` has a local variable called r which is initialized to 0.125 and then it calls `show`.

What does the `main` program do? The `main` program calls `show` and then calls `small` and prints a new line. It does the same thing all over again. Let us trace the values which are printed out by these 2 print statements under static scope and dynamic scope. Under static scoping, `show` prints r ; the only r which is relevant at this point is the r here, which is printed out as 0.25. Then it calls `small`. It has a float variable called r which is initialized to 0.125, but when we call `show` under static scope, again there is no other r except this which is relevant. So, 0.25 is printed all over again. The same goes for the next line – 0.25 and 0.25.

Let's see what happens under dynamic scope. The first `show` - there is only one r , which is active that is this r ; so, it prints 0.25. Then it calls `small`. Now, the second r becomes active because the procedure `small` is called.

The new `r` has 0.125. When `show` is called, the `r` variable that is referred to here is the most recent activation of `r` which is this. (Refer Slide Time: 36:37) So, it prints 0.125. When it comes out and prints a new line and then `show` is called again, at this point this procedure `small` and the variable `r` have vanished; the activation record have already been destroyed because the procedure has returned. 0.25 is printed all over again. `small` repeats exactly what happened. The `small` creates a new variable, it calls `show` and this `r` within `show` is again going to refer to the `r` which is inside `small`. 0.125 is printed all over again. So, this is how the static scope and dynamic scope really work

(Refer Slide Time: 37:18)



Let see how to implement the dynamic scope. Static scope, we already saw how to implement it; that was using the static link dynamic link chain or using the display stack. Here we have what is known as the deep access method and in the next few minutes after this, we will also discuss another method call the shallow access method.

What is the deep access method? In this case, we use the dynamic link itself as the static link. In other words, we do not have 2 links; we just have 1 link. It is dynamic link Static link is not necessary. The reason is simple; we always want the most recent activation of the variable and we are not worried about the scope. So, there is no need to compute the value of static link as we used to do before. Now we just use the dynamic link, but how to find the most recent activation of that particular name. This is done by searching the activation records on the stack; to find the first activation record containing the non local

name. If it is a local name, then it is within my activation record. Whatever I am doing within my procedure activation, the variable is existing.

If it is a global variable, we need to go out, but how far into the stack of activation records is the question. Since we do not know how far, it is called as the deep access method. It can go up to the main program, that is, the first activation record or it could be the immediate previous predecessor activation record itself; we have no idea.

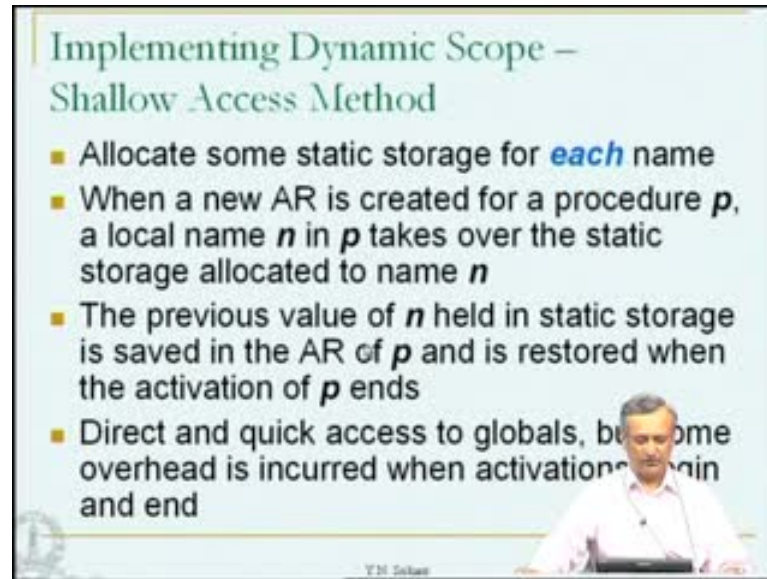
The depth of the search depends on the input to the program and cannot be determined at compile time. This search also needs some information on the identifiers to be maintained at runtime within the activation records. Why?

How do we search for a variable in an activation record? If the activation record has only offsets, there is no way we can say, this is that of the variable r or x or a. So, for each variable, we also need to maintain some type of an integer code saying with this code, this is the variable associated; with this code, this is the variable associated and so on and so forth.

Once we have this coding within the activation record, that is, the variable and the integer code which is already known to the compiler, this can be established by the compiler; that is not a problem. Because each name needs to have a unique code nothing more than that is necessary. We search for the existence of that particular code. Once that code is available, the offset corresponding to it is used for access to that variable.

Such a mechanism takes much longer time to access the globals, but there is no overhead when activations begin and end. This is an advantage. Activations do not cause any overheads, but when you want to access globals, you have to get deep into the stack of activation records and then access the variable.

(Refer Slide Time: 40:49)



Implementing Dynamic Scope – Shallow Access Method

- Allocate some static storage for *each* name
- When a new AR is created for a procedure *p*, a local name *n* in *p* takes over the static storage allocated to name *n*
- The previous value of *n* held in static storage is saved in the AR of *p* and is restored when the activation of *p* ends
- Direct and quick access to globals, but some overhead is incurred when activations begin and end

Y.N. Saha

Let us look at the next method of implementing dynamic scope. This is called as the shallow access method. Why is it called shallow? It will be very clear soon. It is a very simple method, allocates some storage for each name. In other words, if the name has 5 or 6 instances that is, the same name is declared in many procedures it does not matter.

We just want to look at each name and allocate the maximum amount of storage for that particular name. In other words, let us say the variable is *x*; it may be declared as *int* in a particular function or procedure. In another function or procedure, it may be declared as an array; and in third one it could be a struct. What we want is the maximum storage corresponding to this name; it could be the array, it could be the struct, any one of these possible. Of course it can be a single variable, but it can be an *int* but struct is not smaller than *int*; that is, why I said either struct or the array itself.

So much maximum storage is known already and that is the storage which is allocated statically for each name. There is a new unique location in each name, very simple. When a new activation record is created for a procedure *p*, a local name *n* in *p* takes over the static storage allocated to the name *n*. Therefore, there is only 1 location or set of locations corresponding to the variable *n*.

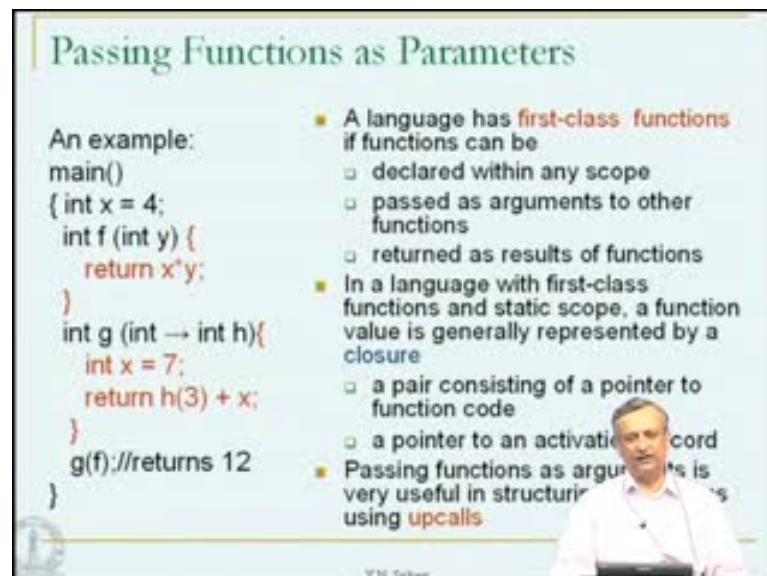
Whenever there is a need for that variable *n*, it could be in a particular activation of a procedure, we just use this static region as the storage allocated to the name *n*. The previous value of *n* is held in static storage As we go along there could be another *n*

which comes along. Then the previous n must vacate the space and give it to the new n. What happens to the value of old n?

The previous value of n held in static storage is saved in the activation record of p and restored when the activation of p ends. Now it is the callee which is storing the variable. Before it tries to access the storage for n, whatever is already available in n is stored in its own activation record. We know the maximum size of that variable storage. That is why we can store it in the activation record. Once the activation of p ends, whatever is in the activation record, the previous value is restored into the static area and then the call continues.

Direct and quick access to globals is possible because this is like hashing. You always come to one location for n; it is static, the address is known, but some overhead is incurred when activations begin and end. You need to store the old value of n and restore it when the procedure p ends. So, this is the overhead that is incurred. Whereas, in the previous case that is, the deep access method there was no overhead when activation begin and end, but you have to traverse deep in to the stack in order to access that particular variable.

(Refer Slide Time: 44:29)



Passing Functions as Parameters

An example:

```
main()
{ int x = 4;
  int f (int y) {
    return x*y;
  }
  int g (int → int h){
    int x = 7;
    return h(3) + x;
  }
  g(f); //returns 12
}
```

- A language has **first-class functions** if functions can be
 - declared within any scope
 - passed as arguments to other functions
 - returned as results of functions
- In a language with first-class functions and static scope, a function value is generally represented by a closure
 - a pair consisting of a pointer to function code
 - a pointer to an activation record
- Passing functions as arguments is very useful in structuring programs using **upcalls**.

Y.N. Srinivas

The next topic that is very important is to understand how to pass functions to other functions as parameters. To understand this, we must understand what exactly first class functions are. Here is a description:

A language has first class functions if functions can be declared within any scope. For example, Pascal has such a facility and functions can be declared within classes in C plus that is, the methods and functions can be passed as arguments to other functions. This is something we have not discussed so far and they can also be returned as results of functions. When we say passed as arguments, we are not looking at a function call as a parameter. It will become very clear with this particular example

Let us look at the example and then return to the text again. Here is main; you have int x equal to 4 and then you have int f which is a function f which returns an integer value, it takes another integer as a parameter and returns x star y, so far so good.

There is another function called g which takes a function as a parameter. The function parameter is h and the function takes as input an integer and returns output as another integer. That is why this is a mapping from int to int.(Refer Slide Time: 46:23) This is the type of this particular h. Just like int was the type of the parameter y here, the type of the parameter here is a map from int to int.

The body contain int x equal to 7, straight forward and then it is says return h 3 plus x which is straight forward again. Let us start looking at the call to g with a parameter f. Notice that, this f is not loaded with any parameters at this point. g has simply an actual parameter called f which is nothing, but this function which is passed as a parameter.

Let us go to g. Types match here; f is int to int, takes input as int and returns output as int. So, int to int is matched here. Now x is initialized to 7. When you say return h 3 plus x that f corresponds to h.

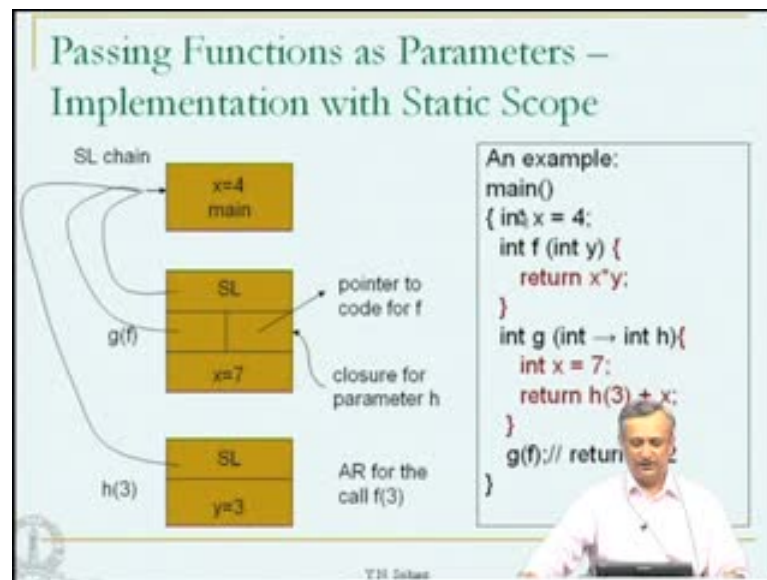
It is really a call to f with the parameter 3. Now, the function f is executed. (Refer Slide Time: 47:32) At this point when we passed f to g, there was no execution of the function f, but here when we call the function h which is nothing, but the function f along with the parameter 3, the actual function f is invoked. That is x star y; it returns x star y. We have 3 and then we have 4 here; so, the return value is 12, plus x again. This 3 star y; g of f returns h of 3; int y was 3; 3 into 4 was 12. This would not return 12. There is a minor mistake here. So, 12 plus 4, 16 is returned here. This should have been 16.

In a language with first class functions and static scope, a function value is generally represented by a closure. We know how to pass integers, floats and so on and so forth as parameters.

Now, let us learn how to pass functions as parameters. It is not enough to pass a pointer to the code itself. We need to pass a pointer to the function code and also a pointer to an activation record. Which activation record? It will become very clear with an example in the next slide.

Passing functions as arguments is very useful in structuring of systems using what are known as up calls. For example, you may have a nested set of procedures. One nested in another and the inner most procedure probably is required to be called at the outer most level which is not possible if you look at static scoping. In order to permit such a thing to happen even under static scope, we can pass the inner most function as a parameter and then execute it whenever it is necessary in the outer function. This is called an up call and it is very useful in operating systems design and implementation.

(Refer Slide Time: 49:44)



Here is an example of closure. Let us explain, what exactly is a closure here. Here is the example that we had seen earlier. You have the main program that is, the variable `x` equal to 4 and nothing else, only one variable. When we call `g` of `f` within the main program, here is the activation record corresponding to that particular call to `g`; there is a static link which points to the main program. We are looking at static scope. The only

things that we can see here from within g are the variables of g itself and the variables of main . Nothing else can be seen that is, we can see x and f that is all.

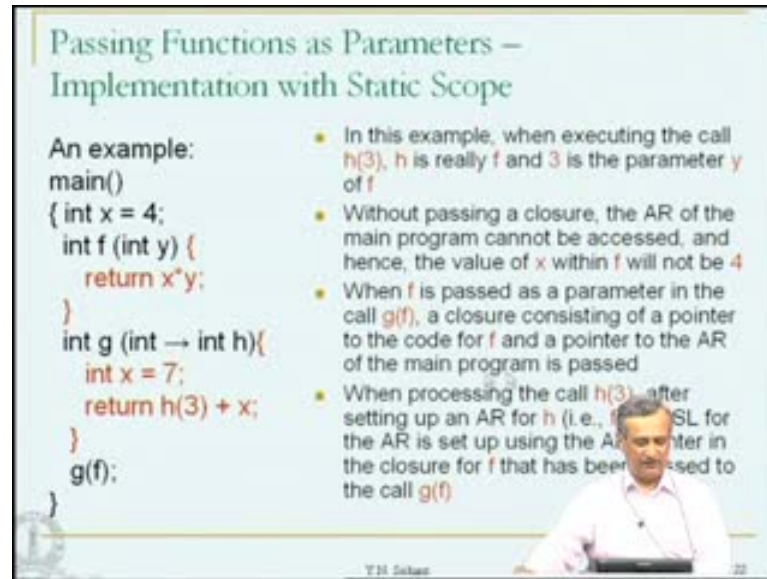
Now here is the integer variable within the function g and the parameter which was passed as f to g . This is the closure corresponding to the parameter h : one part is a pointer to code for f and a second part is a pointer to the enclosing scope that is, the main itself.

When the call to h that is, the call to f is made, the only difference in the implementation is creation of the activation record remains the same, but creation of the static link is actually done by copying this particular static link from the closure parameter to the actual static link variable and there by pointing it to main itself.

This particular activation of f runs as if it is an activation with access to itself and also the activation record of main . This is the only difference. Let me repeat when we create the activation record of the function parameter, the static link is created by copying the static link value from the closure parameter and it is not computed using L_1 minus L_2 plus 1, but when we pass a closure which particular static link to be passed is computed using the same formula L_1 minus L_2 plus one treating f , as if it were a call.

f is at this level. (Refer Slide Time: 52:28) This is let us say 1; this is 2. If you treat it as a call to f , you really have 1 minus 2 plus 1 which is 0, that is, the pointer to the activation record of main itself is to be passed to this particular closure. That is how we determine; it is the same L_1 minus L_2 plus one.

(Refer Slide Time: 52:48)



Passing Functions as Parameters – Implementation with Static Scope

An example:

```
main()
{ int x = 4;
  int f (int y) {
    return x*y;
  }
  int g (int → int h){
    int x = 7;
    return h(3) + x;
  }
  g(f);
}
```

- In this example, when executing the call `h(3)`, `h` is really `f` and `3` is the parameter `y` of `f`
- Without passing a closure, the AR of the main program cannot be accessed, and hence, the value of `x` within `f` will not be `4`
- When `f` is passed as a parameter in the call `g(f)`, a closure consisting of a pointer to the code for `f` and a pointer to the AR of the main program is passed
- When processing the call `h(3)`, after setting up an AR for `h` (i.e., the SL for the AR is set up using the AR pointer in the closure for `f` that has been passed to the call `g(f)`)

Y.H. Sakar 22

Whatever I explained so far is listed here. Just to make sure that we understand everything well let us go through it very quickly. In this example when executing the call `h 3` `h` is really `f` and `3` is the parameter `y` of `f`. This parameter `y` is nothing, but this particular `3`. Without passing a closure, the activation record of the main program cannot be accessed. How do we know what exactly is the scope of this particular `h`? Hence the value of `x` within `f` will not be `4`. We have no idea what this particular `x` will be because we do not even know that we can access the main program.

When `f` is passed as a parameter in the call `g f`, a closure consisting of a pointer to the code for `f` and a pointer to the activation record of main program is passed. This is what I mentioned now. When processing the call `h 3`, after setting up an activation record for `h`, the static link for the activation record is set up using the activation record pointer in the closure for `f` that has been passed to the call `g f`.

(Refer Slide Time: 54:08)

Passing Functions as Parameters – Implementation with Static Scope

An example:

```
main()
{ int x = 4;
  int f (int y) {
    return x*y;
  }
  int g (int → int h){
    int x = 7;
    return h(3) + x;
  }
  g(f); // return 2
}
```

Y.H. Saha

This was what I was saying just now. We really copy this to this, rather than computing this all over again using L_1 minus L_2 plus one. That is really as far as the activation record management is concerned. That is how exactly static scope, dynamic scope etcetera are taken care of.

(Refer Slide Time: 54:17)

Heap Memory Management

- Heap is used for allocating space for objects created at run time
 - For example: nodes of dynamic data structures such as linked lists and trees
- Dynamic memory allocation and deallocation based on the requirements of the program
 - `malloc()` and `free()` in C programs
 - `new()` and `delete()` in C++ programs
 - `new()` and garbage collection in Java programs
- Allocation and deallocation may be *completely manual* (C/C++), *semi-automatic* (Java), or *fully automatic* (Lisp)

Y.H. Saha

Now let us switch to a slightly different topic called as the heap memory management. What is heap? Heap is just an area in memory. There is a difference between heap and stack. Stack is a very systematic data structure; there is only push and there is only a pop

permitted and a stack can be implemented using either link layer star or it can be implemented using an array.

A heap is extremely useful for allocating space for objects created at run time. We do not know when the object is going to be created. Therefore, it is not possible to use the stack; you can only use the heap. For example, nodes of dynamic data structure such as linked list and trees cannot be created using a stack or an array etcetera without losing some flexibility.

Dynamic memory allocation and deallocation based on the requirements of the program. For example, C programs have malloc. They do malloc when they want a node to be created in a link list or in a tree etcetera. They call free when that node is not needed anymore.

Similarly, in C plus plus programs, there is a call to new to create a new object and then there is call to delete the old object. In Java, there is only new that is, a call to new is going to create a new object, but there is no call to delete or free any object. There is a process called garbage collection which collects all the useless objects and returns it to some pool.

Allocation and de-allocation may be completely manual as in C, C plus plus. That is we have malloc, free, new and delete; semiautomatic as in Java that is, we have new and then garbage collection or it could be fully automatic as in the case of a functional programming language such as Lisp.

We will actually end our lecture at this point and in the next class we will discuss more details of how heap is organized. We are going to look at a memory a manager, how the allocation happens, how de-allocation happens, etcetera. Thank you very much.