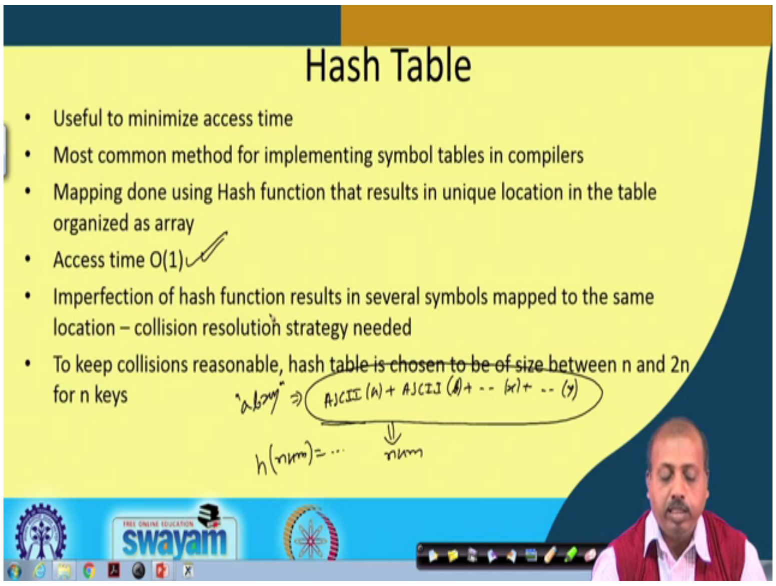
Compiler Design

Prof. Santau Chattopadhyay
Department Electronics and Electrical Communication Engineering
Indian Institute of Technology, Kharagpur

Lecture - 45
Symbol Table (Contd.)

So, hash tables. So, there you used to minimize the access time as we discussed in the last class.

(Refer Slide Time: 00:21)



Hash Table

- Useful to minimize access time
- Most common method for implementing symbol tables in compilers
- Mapping done using Hash function that results in unique location in the table organized as array
- Access time O(1)
- Imperfection of hash function results in several symbols mapped to the same location – collision resolution strategy needed
- To keep collisions reasonable, hash table is chosen to be of size between n and 2n for n keys

$$\text{"abxy"} \Rightarrow \underbrace{\text{ASCII}(a) + \text{ASCII}(b) + \cdots (x) + \cdots (y)}_{num}$$

$$h(num) = \cdots$$

So, this access is a constant time access. So, this is basically access time is order 1. So, as we have noted here this one. So, most common method for implementing symbol tables in compilers. So, that is one we have already seen why is it like that. And mapping done using hash function that results in unique location in that table organized as array. So, there you can think about different hash functions. The if the number if the entries that you are going to put are integers then we can have you can have say hash function which is say like the mod function can be used.

However, if the that where the items that you are going to store in the hash table or on the basis of which will be doing the search, so if it not a number. For example, for symbol table particularly what happens is that we are not going to store them as numbers, but rather we are going to store the name of the symbols as character string and we will be searching based on character string only. Now how to convert this character string to a number? So, there can be several strategies for that a very simple strategy may be like this that, if I have got the got a string like say a b x y ok. Then what we do? We take the corresponding ASCII values of this a b x and y.

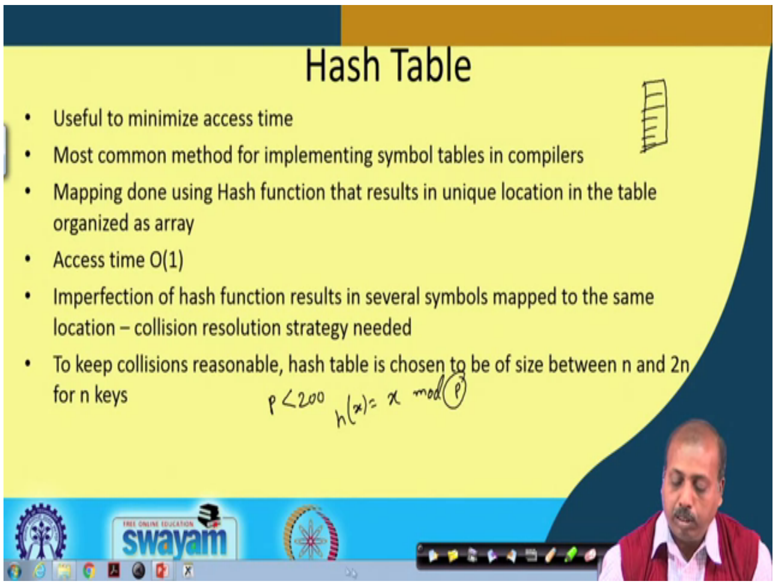
So, we take the ASCII value of a add them. So, ASCII value of b plus ASCII of x plus ASCII of y. So, if we add them then it will give me a number and based on that number I apply a hash function. So, h of number, so it can give me some index of the array index. So, this is just one example. So, is not mandatory that it has to be done like this, but this is just an example because the hash functions that we will consider, so most of the time they will be working with integer arithmetic. So, it is one way to convert one string variable into string name into an integer into an integer value.

So, mapping done using hash function that results in unique location in this, in the table organized as array and access time will order 1. Now imperfection of hash function result in several symbols mapped to the same location. So, this we have discussed in the last class. We have seen that if depending upon the hash function, so several keys may be mapped onto the same location. So, that is called a collision and we have to have some collision resolution strategy.

Now, it has been seen that for normally for this coli to keep the collision reasonable hash table is chosen to be your size between n and 2 n for n keys. So, it is we keep twice the number of keys that we are going to store. So, if you if you expect that the programs that a compiler will compile, will have about say 100 symbols, then you keep the hash table of size about 200.

So, you take a prime number which is just less than to a just less than 200 and then you can just you can apply the hash function.

(Refer Slide Time: 03:51)



Hash Table

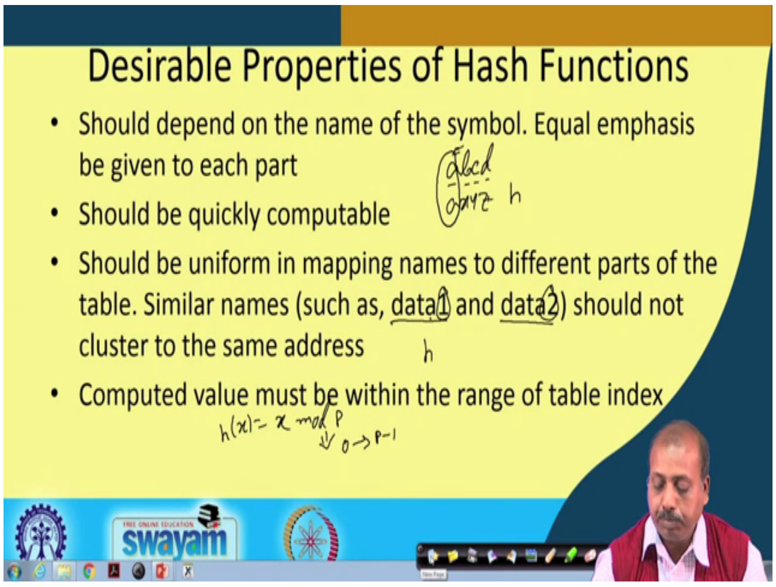
- Useful to minimize access time
- Most common method for implementing symbol tables in compilers
- Mapping done using Hash function that results in unique location in the table organized as array
- Access time O(1)
- Imperfection of hash function results in several symbols mapped to the same location – collision resolution strategy needed
- To keep collisions reasonable, hash table is chosen to be of size between n and 2n for n keys

$P < 200 \quad h(x) = x \bmod \boxed{P}$

Suppose so, I have got a prime number P which is we have suppose I have got a prime number P which is less than 200 and then I can make this h of x to be x mod P. So, in that case though they are only at most 100 symbols, but I am keeping it 200 because then even if there is a collision, so I will be able to so the collision will be less. Because, this P is a large number ok. So, it is expected that the collisions will be less.

So, there are several studies on how to design this hash suppose hash functions and also that these collisions are less and all. So, but that is beyond the scope of our discussion. So, I would suggest that you refer to some data structure classes for that hash table organization. So, for as a compiler designer. So, we are just going to use it for this symbol table organization.

(Refer Slide Time: 04:50)



Desirable Properties of Hash Functions

- Should depend on the name of the symbol. Equal emphasis be given to each part
- Should be quickly computable
- Should be uniform in mapping names to different parts of the table. Similar names (such as, data1 and data2) should not cluster to the same address
- Computed value must be within the range of table index

So, next we will be looking into some desirable properties of this hash function. So, first thing is that it should depend on the name of the symbol and equal emphasis be given to each part.

So, it is like this that I have got the symbol name as say a b c d, I have got a symbol name say a b c d. Now the function that we have the hash function h that I am com computing, it should have equal weight age on all these parts a b c and d. Otherwise what will happen is that, these if 2 names if there is another variable like say a x y z. And if the maximum emphasis is on the symbol a on the first on the first character only of this string, then there is a chance that it will be they will be mapped to the same location by this hash function h.

So, that function h should be such that it gives equal emphasis to each part of the name. Should be quickly computable, so it should not take lot of time to compute the function, because our objective is to reduce the access time, and for all the accesses that we are doing to the table. So, it will be computing this function h.

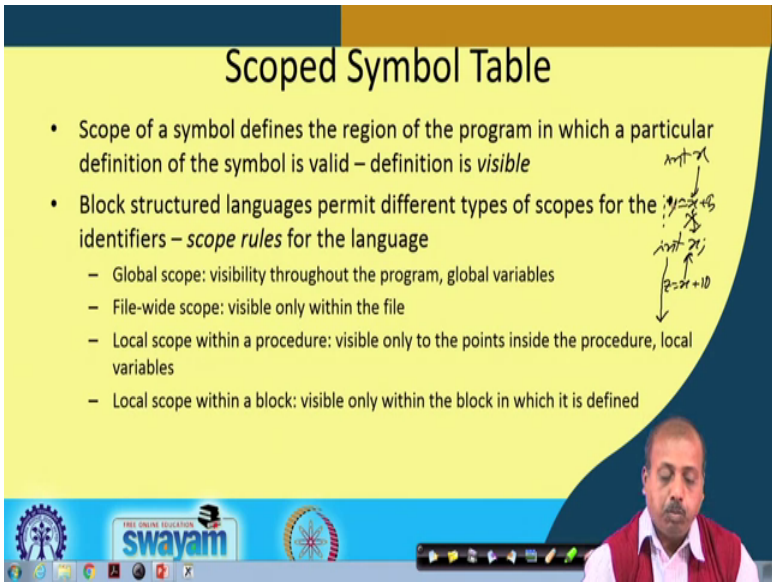
So, it should be suppose quickly computable, so that is another desirable property should be uniform in mapping names to different parts of the table. Similar names should not cluster to the same address. So, it is like this that say this data 1 and data 2 this is an example. We have got 2 variable 2's identifiers data 1 and data 2. Now in these two identifiers only the last say last character is different. So, if I have got an h function the hash function h should be such that this data 1 and data 2, they should be the match to they should be to mapped to so different locations. There should not be matched to the same location.

So, this is known as clustering. So, clustering means if the names are similar. So, there the hash function it generates address which are also similar. So, that should not happen. So, we should we should if the hash function should avoid this clustering. And then computed value must be within the range of table index. So, as I said that we can have this modulo function, so h of x equal to x mod P. So, in this case the range is from 0 to P minus 1.

So, if you are using some other hash function instead of this modulus function which is pretty common. Then what can happen is that, the range of indices that is generated from this h function, so they do not match with the actual table index range ok. So, that way it can create difficulty. So, this is also another desirable property.

So, these are the desirable properties like as a compiler designer, we have to choose the hash function. So, we have to keep in mind all these points to come up with a good hash function.

(Refer Slide Time: 07:53)



Scoped Symbol Table

- Scope of a symbol defines the region of the program in which a particular definition of the symbol is valid – definition is *visible*
- Block structured languages permit different types of scopes for the identifiers – *scope rules* for the language
  - Global scope: visibility throughout the program, global variables
  - File-wide scope: visible only within the file
  - Local scope within a procedure: visible only to the points inside the procedure, local variables
  - Local scope within a block: visible only within the block in which it is defined

So, next we will be looking into; so far we are looking into linear symbol table. So, where there is a single scope and then all the variables and the symbols and identifiers they are visible to the entire program. But it is not the very suppose it is not a very common situation.

So, again most of the programming languages they are hierarchical in nature and at least they have got this block boundaries. So, we have got procedures and functions such that they are independent of each other and whatever variables you are defined within a procedural function. So, this is visible only within that procedural function. It is not visible outside. And similarly, there are some identifiers or some variables which are global in nature. So, that they are visible to the entire program. So, this way we can have different types of visibilities of identifiers in the program.

So, this they for these type of situation for this type of programming languages, so we you should have symbol table which is called as which are known as Scoped Symbol Table. So, this scope of a symbol it defines the region of the program in which a particular definition of the symbol is valid. So, if the definition is or so it is also known as the definition is visible.

So, so if I have got a definition say, in this in this program. So, suppose somewhere here I have got a definition like i n t x. Now, how far in the program this x is visible. So, if I say that this x is visible to the entire program. So, it is visi it is seen, it is visible here. Like, if there is a statement like y equal to x plus 5. So, this x refers to this x. Or it may be so happen that , after this x has been defined from this point onwards only this x is important suppose x x definition is visible.

So, if I have got a statement like z equal to x plus 10, then this x will refer to this, but this x will not in that case because, I said that it is visible from the point from where which it is defined. So, this x will actually correspond to some previously defined x here and this x will correspond to that, so as soon as you are coming to a new definition, so previous definition will be lost. So, that way we can have some scope rules. So, this is a very trivial scope rule that I have talked about.

So, normally programming languages they will define their own scope rule and accordingly we have to we have to come up with different symbol table organizations. So, that we can check those suppose check up the variable and identify definitions following the scope rules.

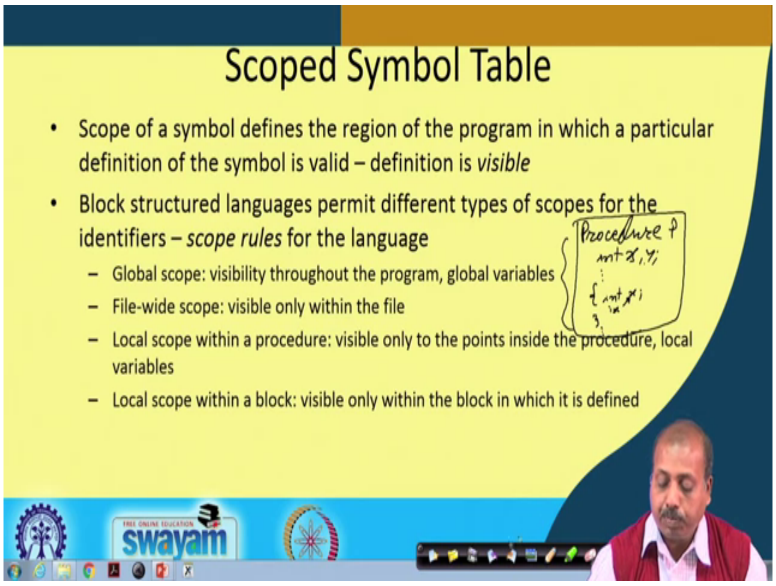
So, this block structured languages, they permit different types of scopes for the identifiers or this is known as scope rules for the language. One is called global scope, and the visibility is throughout the program, so that is the global variables. But almost all the programming languages they will allow these global variables, so they are visible throughout the program. Then there is File-wide scope, so visible only within the file.

So, maybe the same program is distributed over a number of files. May be for may be that there are different suppose people in a group who are developing the software for the same software parts of the software developed by different people in the group and they are defining their own files. So, if I define a variable in my file, so that is local to my procedures.

Similarly, somebody else defines another variable in their in that file so that it is visible only to procedures in that file. So, that is known as File-wide scope. So, this is not much meaningful because ultimately all these files are going to be combined. So, what happens to those combinations, but definitely the code generation and also they will be with respect to this the identifiers is defined within the file at which it is being used.

So, we have the File-wide scope is there. Then there is local scope within a procedure, so in that case it is visible only to the points inside the procedure that is the local variables. Particularly the local variables the scope is the local scope. And local scope within a block it is visible only within the block in which it is defined.

(Refer Slide Time: 12:27)



Scoped Symbol Table

- Scope of a symbol defines the region of the program in which a particular definition of the symbol is valid – definition is *visible*
- Block structured languages permit different types of scopes for the identifiers – *scope rules* for the language
  - Global scope: visibility throughout the program, global variables
  - File-wide scope: visible only within the file
  - Local scope within a procedure: visible only to the points inside the procedure, local variables
  - Local scope within a block: visible only within the block in which it is defined

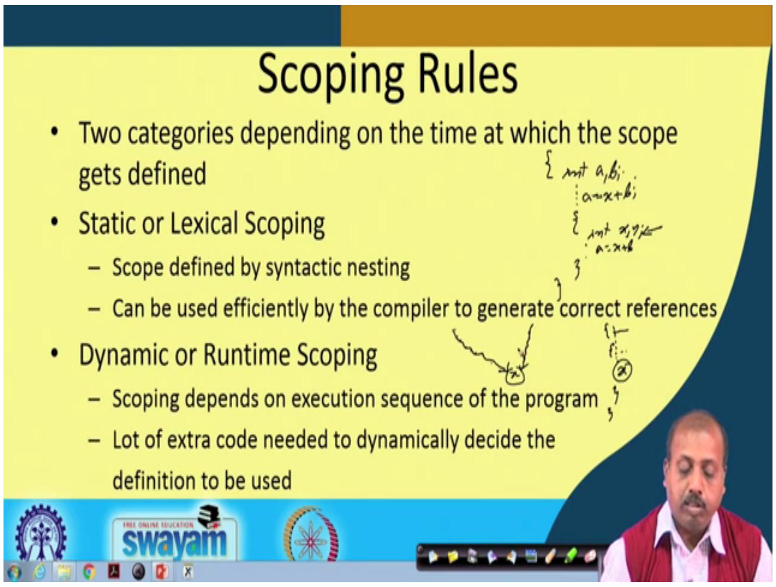
```
Procedure P
  int x, y;
  :
  { int x;
  }
```

So, I can have local scope within a procedure, like I can have a situation like this. So, I have i can have a procedure P and there I can have some variable defines integer x y like that.

So, then this x y, so they are visible to the entire if this is the body of the procedure for this entire body this x and y are visible ; however, it may so happen that in this, there is a block here and within this block another definition of x comes. And so here if I refer to x, then this x will talk about this x. So, this is not that local x for the procedure. So, this is local for the block. So, this is the local scope within a block and. So, this x is not visible outside this block; however, the x that I have here. So, it may be visible everywhere within the procedures.

So, that way we can have certain set of rules for local scope within a procedure and local scope within a block. So, you can have different types of scope rules.

(Refer Slide Time: 13:39)



Scoping Rules

- Two categories depending on the time at which the scope gets defined
- Static or Lexical Scoping
  - Scope defined by syntactic nesting
  - Can be used efficiently by the compiler to generate correct references
- Dynamic or Runtime Scoping
  - Scoping depends on execution sequence of the program
  - Lot of extra code needed to dynamically decide the definition to be used

So, we will be looking into these scope rules and how are they going to affect this programming language converter suppose code generation for different programming languages. So, there are 2 categories of scoping rules on that on based on that time at which the scope gets defined.

So, one is called static or lexical scoping another is called dynamic or runtime scoping suppose. So, this static or lexical scoping scope is defined by syntactic nesting. So, if I have got something like this, suppose there is a block and , suppose there is a block and in that block I have got this suppose variables a a and b. And then, suppose so there is another block here and there I have got the variable integer x y here.

So, this syntac static or lexical scoping tells that as you are going by into this. So, whatever is defined in the outer block, so it is also visible to the inner block. But whatever is defined in the inner block, so that is not visible to the outer block. So, this x and y variables are not available at this point.

So, if I write something like x plus b something like that, so that that in that case compiler should give the error that x is undefined, because x is not defined in the current scope or the local scope of the this at this point. However, if I write say a equal to x plus b at this point, so that will be absolutely fine because this a and b so they will be coming from the outer block. So, from the outer block these variables are identifiers are defined and they are used they can be used in the inner block.

So, this is the scope defined by syntactic nesting. So, be using this nesting, we can define this scope. Now they can be used efficiently by the compiler to generate correct references. So, this is a static or lexical scoping, so this is much more easier for the compiler to handle, because it can check the situation and accordingly it can generate the references. Then, we have got dynamic or runtime scoping. So, here the scoping depends on the execution of sequence of the program.

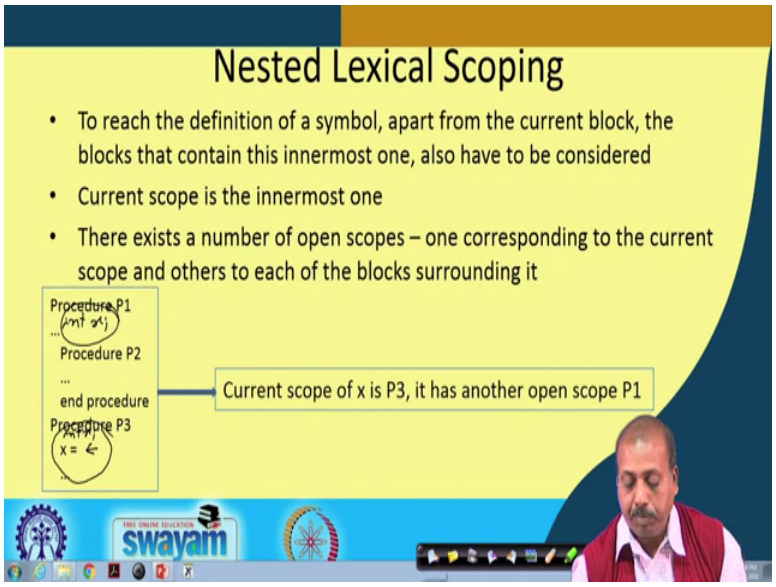
So, naturally there are lot of extra code needed to be dynamically decide the definition to be used. So, basically in this case it depends on say, how we have we reached a particular point. So, it is so previously. If this is a program and at this point of time, so if I have got a reference to a variable x ok. So, if it is a static or lexical scoping, so I just need to check the blocks. So, this blocks.

So, if I do not find x defined here I need to look into the surrounding block for x ok. So, that way, it can find the x value variable. However, in case of dynamic scoping, so it depends on the sequence by which we have come to this point. So, it, so may be that we have come through this path. So, accordingly this x that we are talking about will be we will have to search for x onto this path. Similar in some other execution if we have come to x where this path in the program then we have to search for into these procedures for a definition of x.

So, that way it is statically we cannot determine it. So, it has to be done dynamically. Only at runtime it will be say for as it will be may looking into this h differences, the different paths by which we have come to x this particular point and accordingly it will be finding out the correct reference. So, this dynamic of runtime scoping so that is there which is in some programming languages, you will find it, but of course, it is a very complex concept to implement ok.

So, this if we have to support this in the compiler generator it has to generate lot of extra code so that it can dynamically find out the reference during execution.

(Refer Slide Time: 17:46)



Nested Lexical Scoping

- To reach the definition of a symbol, apart from the current block, the blocks that contain this innermost one, also have to be considered
- Current scope is the innermost one
- There exists a number of open scopes – one corresponding to the current scope and others to each of the blocks surrounding it

```
Procedure P1
  int x;
...
  Procedure P2
  ...
  end procedure
Procedure P3
  x =
...
```

Current scope of x is P3, it has another open scope P1

So, nested lexical scoping. So to reach the definition of a symbol apart from the current block, the blocks that contain this innermost one also have to be considered. So, so suppose this is the situation. So, let us take this example where we have got a procedure P 1 and within this procedure P 1 another procedure P 2 has been defined and this procedure ends at this point. Then, we have got a procedure P 3 and in P 3, we have got a reference to x.

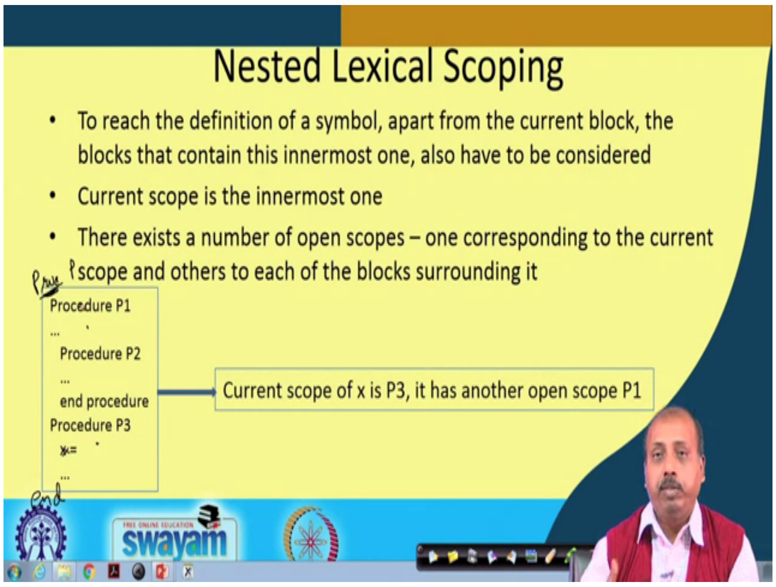
Now, where to look for this x? So, this procedure P 3, you see that it is nested within procedure P 1. Similar procedure P 2 was also nested within P 1, but procedure P 2 is over. So, when we are at this point, we have got 2 possible scopes for x. One possibility is that here itself in the procedure P 3 itself x is defined, so that is one possibility. And other possibility is that, it is defined in the procedure P 1, so here somewhere here it is defined.

So, first it will look into this procedure P 3 definitions. So, if it can find this x within that, so it will be using that reference. And if it is not, then it will be looking into this P 1 scope because P 1 is the next higher level of nesting that we have. So, if we look into P 1's scope and accordingly it will try to find the reference.

So, to reach the definition of a to reach the definition of a symbol apart from the current block , the blocks that contain this innermost one also have to be considered. So, it is not that we look only in procedure P 3, but we also look into the other surrounding blocks.

So, current scope is the innermost one and there exists a number of open scopes one corresponding to the current scope and others to each of the blocks surrounding it.

(Refer Slide Time: 19:48)



Nested Lexical Scoping

- To reach the definition of a symbol, apart from the current block, the blocks that contain this innermost one, also have to be considered
- Current scope is the innermost one
- There exists a number of open scopes – one corresponding to the current scope and others to each of the blocks surrounding it

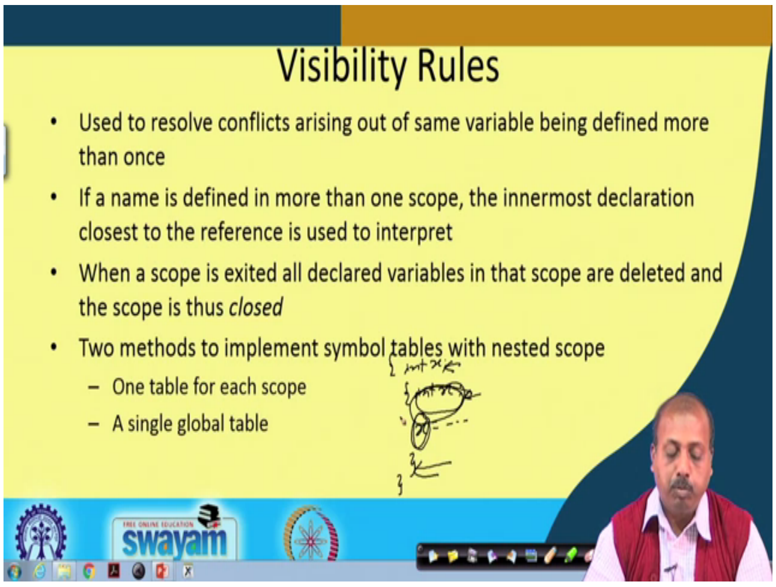
```
Proc
Procedure P1
...
  Procedure P2
  ...
  end procedure
Procedure P3
  x=
  ...
End
```

Current scope of x is P3, it has another open scope P1

So, if there is another block surrounding it, so if this procedure P 1 was inside another procedure say P, and the procedure P ends here. Then this x so when a when you are talking about this x it has got an open scope x, a open so open scope P 3 P 1 as well as another open scope P. So, it will have 3 scopes in that case P 3, P 1 and P.

So, that way hierarchically it has to go for from one nesting level to the previous nesting level. So, till it comes to the outermost level, so all these pros all these nesting levels are available for probable scope.

(Refer Slide Time: 20:30)



Visibility Rules

- Used to resolve conflicts arising out of same variable being defined more than once
- If a name is defined in more than one scope, the innermost declaration closest to the reference is used to interpret
- When a scope is exited all declared variables in that scope are deleted and the scope is thus *closed*
- Two methods to implement symbol tables with nested scope
  - One table for each scope
  - A single global table

So, in this way the nested lexical scoping is defined. And there are some visibility rules. It says that, it is that these are used to resolve conflicts arising out of same variable being defined more than once. And if a name is defined in more than one scope, the innermost scope the innermost declaration closest to the reference is used to interpret. So, this is actually the tiebreaking rule like it may so happen that we have got this thing. We have got say, we have got say 1 the x equal to something here and so this x is defined in this block as well as it is defined in some outer block.

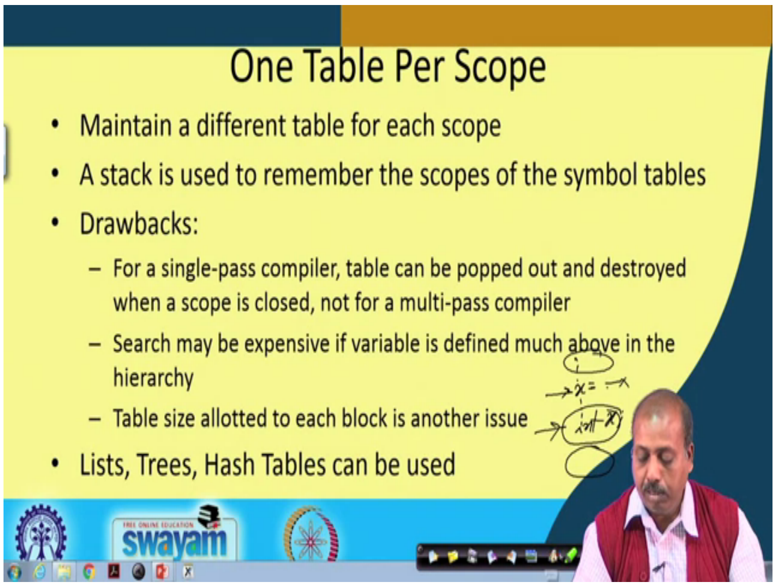
So, this is also defined here, in some outer block. So, in that case, this its the rule says that for getting reference so it will refer to the local one. So, to the innermost one and if it is not available innermost one then only this outer one will be used.

So, that is the innermost declaration closest to the reference it will be used for the interpretation. And when a scope is exited, all declared variables in that scope will be deleted and the scope will be closed. So, this scope is exiting like say one procedure compilation is over. So, there, so all the local variables for that procedure so they are no more visible, so as a result they can be closed and we can have also this type of situation that one. So, we have compiling and one block is over. So, this is the we have we have we come to the end of a block.

So, at that point also. So, all the symbols that are defined in this block , so they can be wiped off and. So, that is the situation. So, that is called the closing of the scope. The scope is thus closed.

There are 2 methods to implement symbol tables within a state scope. 1 is 1 table for each scope, another is a single global table. So, both the approaches are used in compiler design with they have got their relative merits and demerits.

(Refer Slide Time: 22:35)

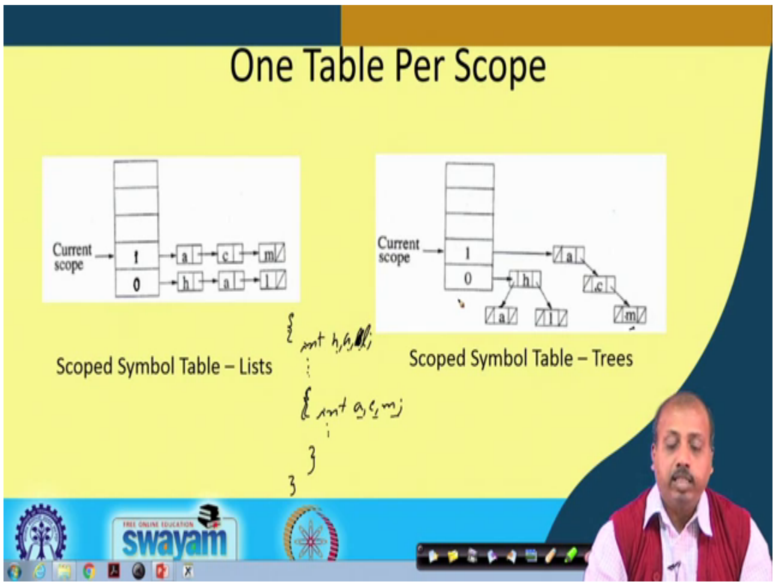


One Table Per Scope

- Maintain a different table for each scope
- A stack is used to remember the scopes of the symbol tables
- Drawbacks:
  - For a single-pass compiler, table can be popped out and destroyed when a scope is closed, not for a multi-pass compiler
  - Search may be expensive if variable is defined much above in the hierarchy
  - Table size allotted to each block is another issue
- Lists, Trees, Hash Tables can be used

So, 1 table per scope it says that, you maintain a different table for each scope and a stack will be used to remember the scopes of the of the symbol tables.

(Refer Slide Time: 22:50)



One Table Per Scope

So,. So, naturally, so we have got its got an example like say we have got a stack where this is the symbol table. Symbol tables are organized in the form of a list, so this is h and a. So, these are two symbols at h a and l. So, these are three symbols in the scope 0. So, this example that we are talking about it is something like this say, we have got this say integer h a l.

And sometime later, we have got another brace where we have got integer a c m. Now for this, so when I am, when I am in this particular block. So, I had to create a symbol table where this h a and l, so they are put on a list. And we use a stack, so we will note down the level of the nesting and the nesting level is 0. So, this h a and l are there. Then, after that we find this another brace. So, the nesting level becomes equal to 1.

So, a nesting level is implemented, it becomes 1. And then we find this symbols a c and m, so they are put on a list so making the symbol. ma making the symbol table for this one. And there, this nesting level is the nesting level is equal to 1. So, that is noted here. So, this is the situation one table per scope under linear table organiser under list type of organization.

So, you can have 3 type of organization also like, so individual tables that we have got. So, they are organized as trees like this h a l. So, they are organized as a tree. So, h is at the root, so a is less than h. So, it is on the left side and l is more than h, so that is on the right side.

Similarly, this a c m, so is at the root, c is greater than that and m is further greater than that. So, that forms a table, so accordingly this stack that we have so they contain the scope definitions. The current scopes, so they it is the scopes 0 and 1. And there is a current scope pointer that points to the current scope that we have. So, this way we can think about this scope symbol table.

So, , so we can have, we can maintain a different table for each scope and a stack will be used to remember the scopes of the current say of the scopes of the symbol table. So, when the current procedure or current function is over. So, we can just get rid of these thing like if this current scope is over, then this table can be deleted and this stack pointer can be decremented. This current scope pointer can be decremented. So, that it points to the next innermost scope.

So that way, we can do this thing. So, the there are some drawbacks of this particularly for single pass compiler. So, table can be popped out and destroyed when a scope is closed, but not for a multi pass compiler. So, for a single pass compiler means that creates the symbol table and does code generation at the same phase so or single in a single pass it does both.

But there are some compilers which now which do it in 2 phases. In the first phase, it makes the symbol tables calculates all offsets and all for all the symbols. So, that code generation part becomes simple. In particular, what happens is that in if in a programming language, we have got the we have got this type of facilities that I can refer to some variable x here and this x will be defined sometime later in the program.

So, that in that case, if I have got a single pass compiler then the difficulty is when you are here. So, you do not know the size of x ok. So, that way you cannot generate a proper code. So, in a two pass compiler what will happen is that, in the initial in the first stage, so it will ignore all these statements. So, it will just look for these type of definitions in the program. And accordingly, it will make the symbol table. It will definitely calculate the offsets like getting these type of statement, so it will calculate the size of those statements and it will accordingly come it will compute the offsets of all the statements. So, that it when it comes to this line. So, it knows it knows the offset of x.

So, that way at the end of phase one, so we have got only the symbol table constructed and in phase two the actual code generation will be done. So, that is a multi-pass compiler. So, in case of single-pass compiler, as soon as we are through with a procedures, so we can forgive forget the corresponding symbol table. So, the it can be popped out and destroyed. However, multi-pass compiler that is not the case, because now we have to remember all the individual compiler individual symbol tables that we had made.

So as a result, search may be expensive if variable is defined much above in the hierarchy. So, because now we have to search number of tables are to be searched. Like you see, that in this table, in this table say in this table. So, if you are looking for say the symbol h ok. Then, first if you look into the current scope so, it will scan through all this one and then it will come it will not find h here, so it will come to this level 0 and then it will be searching for this h. So, it will get, it will be getting h here.

So, in the worst case when the symbol is not present in that, in any of these tables then what will happen? The all the table entries are to be searched to tell what is or where is it. So, that way, it is difficult because the overhead becomes more. So, it has to search through a number of a number of places ok. The number of tables for getting the offset.

So, search is search may be expensive. Table size allotted to each block is another issue, because how big a table you will be assigned to individual blocks. So, if you are if the size is not is not estimated properly, then we may either can give it a small number or a large number of cells as a result the table may be underutilized or it may be that table is not sufficient. So, all these data structures like list trees and hash tables can be used for this one table per scope type of organization.