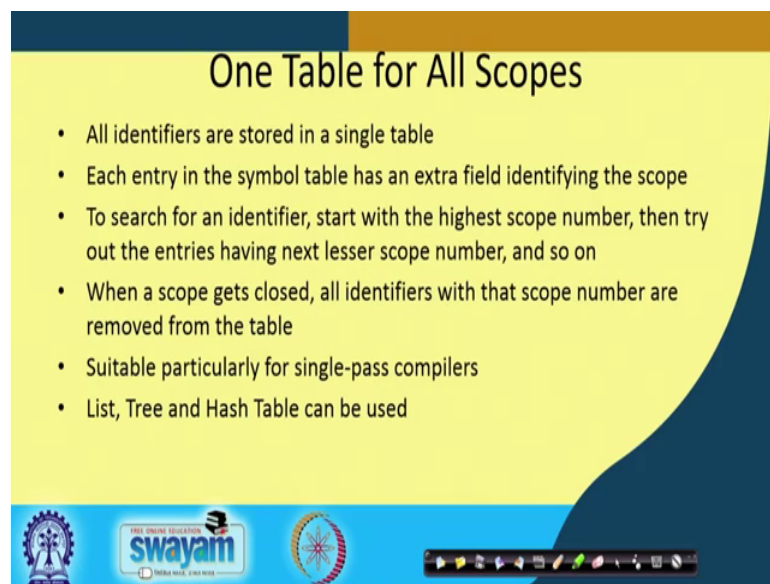


Compiler Design
Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

Lecture - 46
Symbol Table Runtime Environment

This one table per all scope type of organization; so, we can have different table organizations as we have seen previously.

(Refer Slide Time: 00:23)



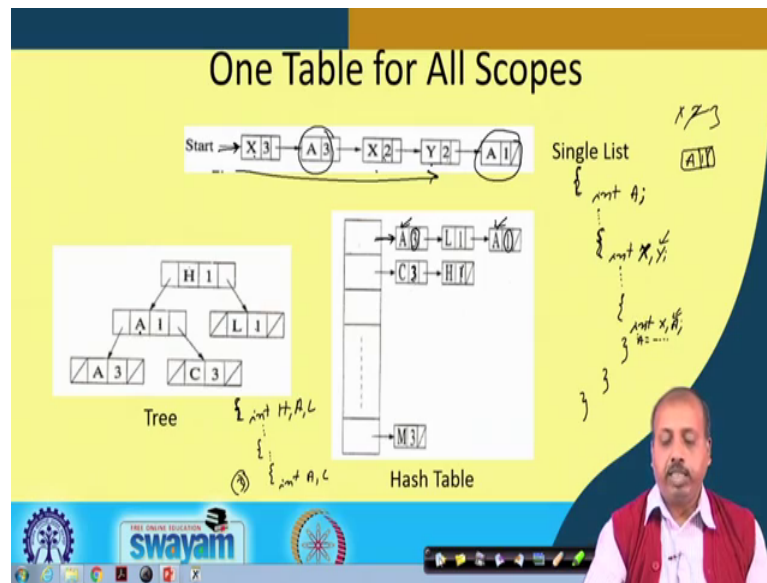
One Table for All Scopes

- All identifiers are stored in a single table
- Each entry in the symbol table has an extra field identifying the scope
- To search for an identifier, start with the highest scope number, then try out the entries having next lesser scope number, and so on
- When a scope gets closed, all identifiers with that scope number are removed from the table
- Suitable particularly for single-pass compilers
- List, Tree and Hash Table can be used

The slide features a yellow background with a dark blue curved shape on the right side. At the bottom, there are logos for IIT Kharagpur, Swayam, and a circular logo, along with a video player control bar.

Like it can be an array, it can be a list, it can be a tree, or hash table, so any of them can be used. So, we will see that how they can be used for that purpose. So, all identifiers are, so if I have got one possibility is that we have got one table for all scopes. So, we have got a single table and that suffices, so that is the made for all scopes together. So, all identifiers will be stored in a single table ok. And each entry in the single table has an extra field identifying the scope.

(Refer Slide Time: 00:57)



So, let us take an example and then try to see. Like here, if I am having a list type of organization of the symbol table; so this is X A X Y A. So, these are some symbols defined in the program and we are storing the corresponding levels. So, the scope number, so in this particular example, so the situation is like, this that we have got one outer block where I have got this integer A. And then within that, so we have got another block where it is this X and Y are defined, integer X x and y. And that after that, there is another block within this where this A and X they are defined; integer X and A, so they are defined; so, this is the nesting.

So, what happens is that, when you have got this one. So, at this point, so this is, so previously my level was 0. Now this, I have found this integer A and, so as soon as we find this brace; so level is incremented by 1. So, this scope level the current scope is made equal to 1. With that it comes to this point and it gets a symbol A. So, this A is put into the symbol table A and it is the corresponding scope number is also noted. So, that is why it is A and 1.

So, this block is this box is created you can say in the symbol 2. So, it has got only 1 entry now. Then it goes forward. Then after some time it finds that there is another brace; so, this current scope is incremented to 2. And then it finds there is 2 new definitions X and Y, so they are put into the symbol table like this and then, their corresponding types are corresponding scope levels are 2. So, these 2 this a scope number is stored there.

Then after some time it comes to another brace. So, this scope is incremented to 3 and this X and A. So, it that there will be referring to this X and A; so they will be stored here this X and A.

Now in this code, so if there is a reference to say A equal to something. In that case, it will start looking from this point and it will try to see where this A occurs first. So, the A occurs first at this point, so it will be, it is referring to this particular A.

Similarly, if it is referring to Y, then it starts from this point searches for Y and it gets the Y at this point; so it is taken as Y. So, that the so it is taken as this Y at level 2. So, this way we can have one table for all scope. Similarly for a tree type of organization, so it may so happen that that at this H A and L. So, they are, so we have got these symbols H A and L in the top level. So, at this level we have got say integer H A and L. Then after sometime we have a brace here there is no declaration, but after some time again there is a brace and there I have got this integer A C.

So it may be like this and then we can see that. So, this, so as it is coming to the third brace. So, this at this point the scope number is equal to 3. So, in tree organization also well apart, apart from the name of the symbol so we are also storing the corresponding scope number. So, these are 1 for H A and L and 3 for A and C.

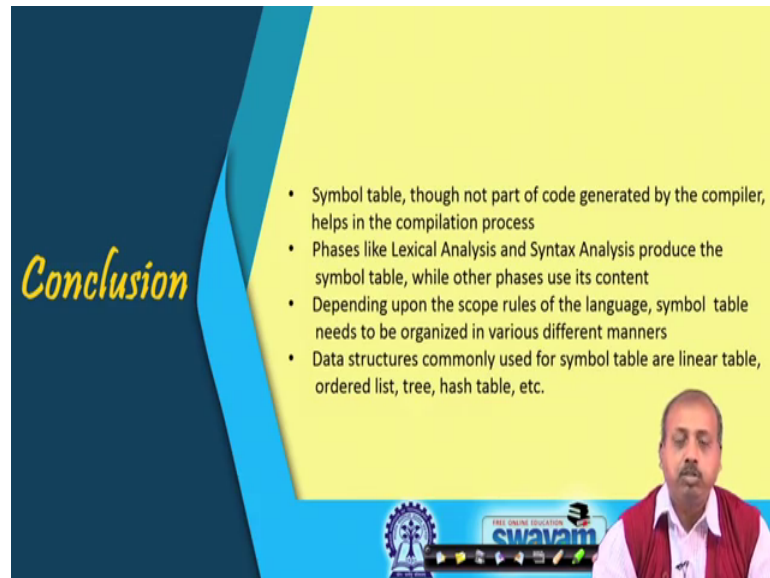
So, you can also have a hash table organization; so, otherwise it is remain same. So, this key values A L C so they are used for doing the hash for applying on the hash function. So, the hash function is so since these two, since these two identifiers both are A their names are A. So, they are matched to the same location. So, we are using a chaining type of approach for this collision dissolution and you also store the corresponding levels ok; so corresponding scope level that is 3 and 1.

L is also mapped here because my it is assumed a it may be the situation the hash function is such that that A and L they map to the same location. Similarly C and H they map to the same location. So, this C, so we also store the corresponding level, the scope level 3 and 1 with C and H and similarly at this point for M stored.

So, in this way we can have a single table for all scopes and that way we can organize the symbol table ok. So, the advantage is that we do not have to have many many symbol tables and it is at a common place you can say. Now whether it is convenient or not, so

that is dependent on the compiler designer, so there is not much choice in fact. So it is up to the compiler designer to decide like which one is more comfortable with him.

(Refer Slide Time: 06:24)



Conclusion

- Symbol table, though not part of code generated by the compiler, helps in the compilation process
- Phases like Lexical Analysis and Syntax Analysis produce the symbol table, while other phases use its content
- Depending upon the scope rules of the language, symbol table needs to be organized in various different manners
- Data structures commonly used for symbol table are linear table, ordered list, tree, hash table, etc.

swam

The slide features a dark blue background on the left with the word 'Conclusion' in a yellow, stylized font. The right side has a yellow background with a list of four bullet points. A video inset in the bottom right corner shows a man with a beard and glasses, wearing a red vest over a white shirt, speaking. At the bottom center, there is a logo for 'swam' with the text 'FREE ONLINE EDUCATION' above it.

So, next we will be looking into the come to the conclusion of this step this discussion.

So, symbol table, so though not part of code generated by the compiler, it helps in the compilation process. And phases like lexical analysis and syntax analysis produce the symbol table; while other phases use its content. And depending upon the scope rules of the language, symbol table needs to be organized in various different form different manners.

Data structures commonly used a linear table, ordered list, tree, hash table etcetera. So, and as I said that there can be other data structures also and we can go for those data structures. So, depending upon the programming language, so the compiler designer may use some innovative data structures. And this is entirely up to the compiler designer. So, with just by consulting the lexical rule or scoping rules of the language, can come up with different type of organization for the tables.

So, we complete this part on this discussion.

So, next we will be looking into the; this how can we organize the runtime environment for this compiler; for in the code generated. So, runtime environment what it means is that, while running a program we need to maintain certain information. For example, you

are calling a procedure. So, when a procedure is being called we need to pass a number of parameters. Similarly when we are returning from the procedure then the return address and the return value, so those are important things, so they are to be taken into consideration and they should be put into the proper places.

So, these are some important points and then this, in this runtime environment management part. So, it will do something, so that the compiler puts sufficient amount of code for this runtime management of the system.

(Refer Slide Time: 08:25)

CONCEPTS COVERED

- ❑ What is Runtime Environment
- ❑ Activation Record
- ❑ Environment without Local Procedures
- ❑ Environment with Local Procedures
- ❑ Display
- ❑ Conclusion

Nested Procedure Call
Definition

```
R() {  
    call q();  
}
```

So, the topics that we are going to cover are like this. That, what is runtime environment? Then there is a concept called activation record. So, activation record is a collection of information which are important for this procedure call and return.

And we will see that there is well defined set of rules between the procedure which is calling a sub procedure. And similarly, there is also some part of some responsibilities on the called procedure, both at the time of going to the going to the procedure and coming back from the procedure. So, there are well defined actions to be taken by these individual modules.

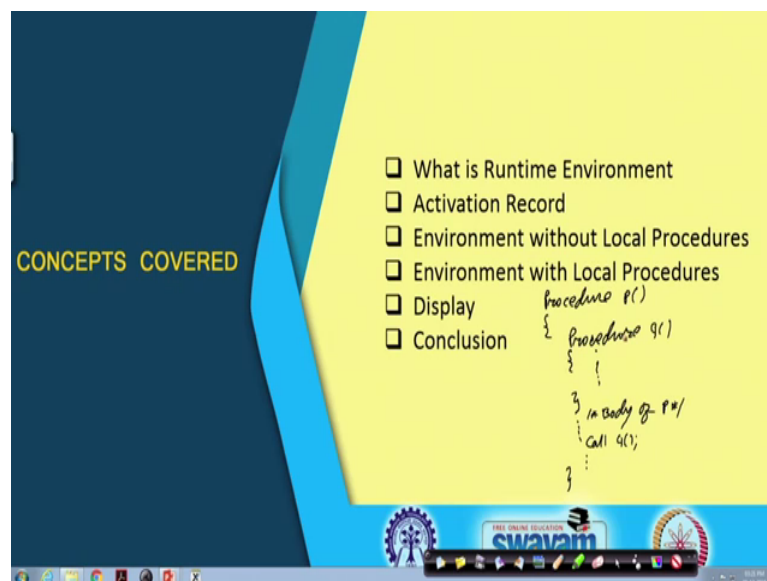
So, we can have environment without local procedures and there can be environment with local procedures. Some programming languages are such that we apply it is

allowing us to have nested definition of procedures. So, it is there are two situation, one is called nested procedure call another is nested procedure definition.

So, one is nested procedure call another is nested procedure definition. So, they so, when I say nested procedure call; so, this is very common. So, almost all programming languages they will allow you to do something like this that. Suppose this is a procedure P, so this is a procedure P and then from this body of the procedure you call another procedure q, so that is the procedure that is the nesting of procedure. So, or nest a nested call of procedure. So, from P, so from the main program you have given a call to P and while executing P it is giving a call to q. So, the q is called within the procedure P. So, that is the nesting of the procedure. Nesting nested call of the procedure.

Now, it may so happen that in the procedure P I define another procedure.

(Refer Slide Time: 10:38)



So, I can have say I have got a procedure P and within this we define another procedure q. So, we define another procedure q. So, this is the procedure q that we have. And after that, the body of P starts. So, from this point I can say this is the body of P. Now here you can have a call to q. You can have a call to q, but can I have a call to q from outside the procedure P?

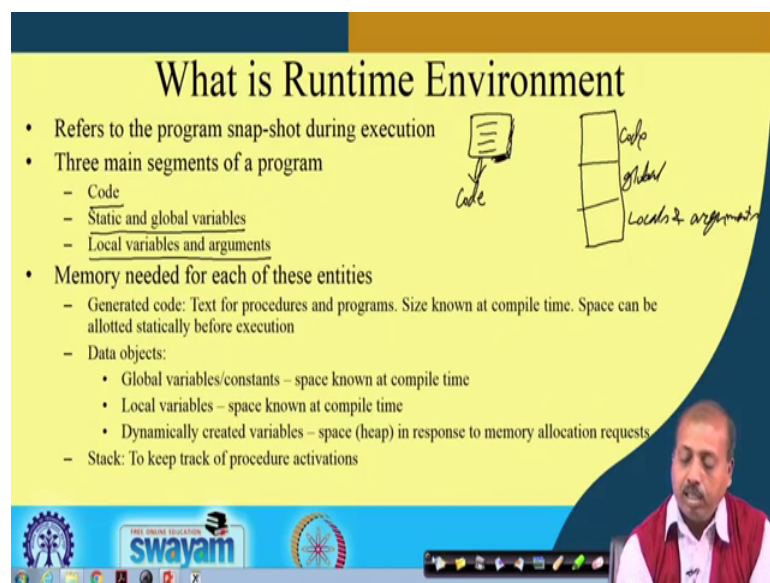
So, some programming languages still that you can do that. Some programming languages will tell you, no you cannot do that. Some programming languages will even

say that you have to call procedure P once before you are going to call procedure q; so, that is also there. So, different programming languages they have different types of semantics for these local procedures.

So, this q is a procedure which is local to the procedure P, so this runtime environment for procedures with for situation with local procedure is different from situation without local procedure. So, naturally, with local procedure the situation is going to be difficult, because all these scope rules of the language, so they will come into play. And accordingly they will be giving the types of identifiers and these types of procedures and all that is there.

Then there is a concept called display. So, displays for making this runtime execution faster ok, so how to do it fast? So, that will be having some special data structure called display. So, that is put into the runtime environment, so that it will be doing it first. So, we will see all these concepts and finally, we will draw conclusion on this thing.

(Refer Slide Time: 12:43)



What is Runtime Environment

- Refers to the program snap-shot during execution
- Three main segments of a program
 - Code
 - Static and global variables
 - Local variables and arguments
- Memory needed for each of these entities
 - Generated code: Text for procedures and programs. Size known at compile time. Space can be allotted statically before execution
 - Data objects:
 - Global variables/constants – space known at compile time
 - Local variables – space known at compile time
 - Dynamically created variables – space (heap) in response to memory allocation requests
 - Stack: To keep track of procedure activations

The slide includes two hand-drawn diagrams. The first diagram shows a box labeled 'Code' with an arrow pointing to it from the word 'Code' written below. The second diagram shows a vertical stack of three boxes. The top box is labeled 'Code', the middle box is labeled 'Global', and the bottom box is labeled 'Local & arguments'.

So, how are you going to go through the discussion? So, first we try to answer this question that, what is runtime environment? So, runtime environment it refers to the program snap-shot during execution.

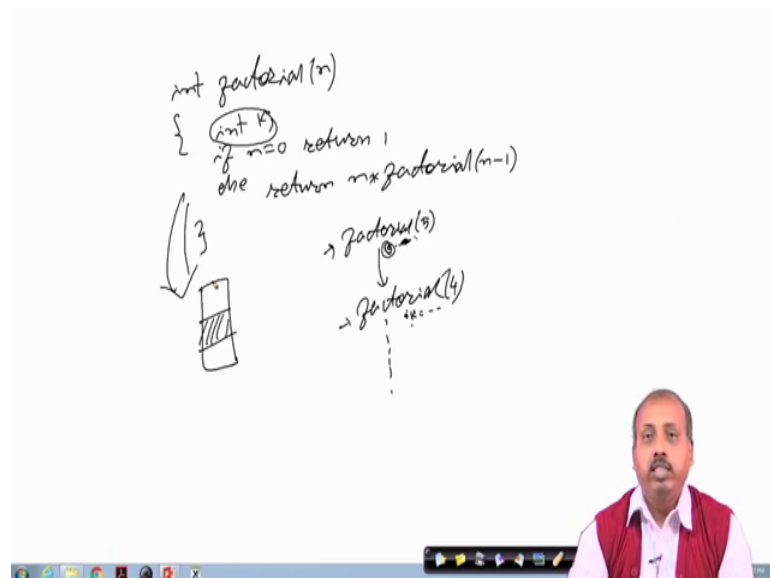
So, program snap-shot means, the program is running and as if I take a snap-shot of the program as it is running. Now you see that a program may be very small, when it is

residing in disk; so, it may be the object code that is produced is it may be very small. So, it may have only a few lines. But when it is executing in the memory in the computer, so then this; the snap-shot may be very large. Why? Because any program that you have, so it has got three portions in it.

So, this body if you look into so it can be divided into three parts, one part of it is called code, another part is called the static or global variables, and another is local variables and argument. So, these are the three parts. If you look into this piece of program, so you can find that there are there so the there so it is the code. But when you take the snap-shot then this snap-shot can be divided into 3 portions. Some part of it is code, some part is corresponding to the global variables, and some part of it is corresponding to the locals and arguments.

So, the static variables, so they are also similar to global variables only, because they will be giving they are treated in a global fashion only, so they are treated; so, static variables and global variables, so they are taken together. Now what can happen is that, suppose I am writing a function which is say suppose I am writing a function which is say factorial function.

(Refer Slide Time: 14:42)



So, this factorial function, so I can write it like this. Int factorial in and then I can say if n equal to 0 then return 1 else return n into factorial n minus 1.

So, this is the piece of program that I have. Now suppose, so when this is there in that disk as the object file. So, I have got the translated version of this program here. So, this is the translated version. And here, so now, what will happen is that when this program is in execution, then what happens is that; suppose I have given a call to factorial 5. So, this factorial 5 has in turn given rise to factorial 4 ok. So, it has given rise factor then factorial 3 like that.

So, what happens is that at a runtime you can understand there can be several instances of this function that are available. Where when it was in the disk you see that I have got only a part of the code which corresponds to this factorial function. But while I am executing the program, so there are several copies of the factorial program that are running. So, naturally, so they I they are not same because the variables that this function is computing are not same as the variables that this function is computing.

The situation will more become more difficult if I have some local variable say K here. Then the K that I am referring to in this function factorial 5 call, so that K and the K that I have in factorial 4, so this K, so these two K's are not same. So, they must be two different K's. Otherwise if this factorial 4 functions modifies this K; so then this K will get affected, so that is not desirable. So, this all the local variables that I have, so there has to be separate copies for them, so, this is actually the challenge.

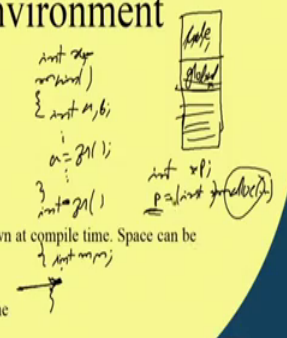
So, whenever you have got this execution going on. So, you get a snap-shot of the program and this program snap-shot is not the same as the original say object file copy that you have. And then there are some, so there are more things that we have then how are you going to manage this situation. Like how are you going to have so many invocations of this same function. Or if there is a nested call, then how are you going to take care of the nested call. So, that is that is the challenge of this particular part of discussion.

So, this refers to memory snap-shot during execution there are 3 main segments of a program, the code segment, static and global variables, and local variables and arguments.

(Refer Slide Time: 17:53)

What is Runtime Environment

- Refers to the program snap-shot during execution
- Three main segments of a program
 - Code
 - Static and global variables
 - Local variables and arguments
- Memory needed for each of these entities
 - Generated code: Text for procedures and programs. Size known at compile time. Space can be allotted statically before execution
 - Data objects:
 - Global variables/constants – space known at compile time
 - Local variables – space known at compile time
 - Dynamically created variables – space (heap) in response to memory allocation requests
 - Stack: To keep track of procedure activations



```
int x;  
main()  
{  
  int a, b;  
  a = 20;  
  int p1();  
  int p1()  
{  
  int m, n;  
  ...  
}
```

The diagram shows a memory stack with segments labeled 'code', 'global', and 'local'. The 'local' segment is further divided into 'main' and 'p1()' (the active function call). The 'main' segment contains 'int x;'. The 'p1()' segment contains 'int m, n;' and 'int p1()'.

So, if I have got a C program, so suppose I have got something like this. That, I have got an integer variable x and then, so this is the C program only, then there is a main function. And in the main function, I have got some local variables a b etcetera. And it is some some point of time it is giving a call to function f 1 and here I have got the function f 1. And there I have got the local variables and also int m n like that.

So, if you take the snap-shot of this program; suppose at when the program is executing at present a, if the main routine has given a call to f 1 and we are at this point of execution of this function f 1. Then, I can say that the program that I have in the memory now, in the main memory that the snap-shot of the program that I have now.

So, it can be divided into 3 parts. One is the code part, code part is actually the translated version of all these statements that I have. All these high level statements, so if you translate into machine code. So, what is it? And the code part is not going to change; so, it is going to be static in nature. So, it is not going to change. Like then at this point I have got a variable x, so that comes to the global variables section. So, we have got this code, then we have got the global variables. So, this variable x will be assigned some space on to this section. And all these variables like a b then m n, so they will come as local variables and that local variable are to be handled and variable and arguments are to be handled.

So, this way any may any program while it is executing it can be thought about consisting of 3 segments, code, static, and global variables, and local variables, and arguments. And memory needed for each of these entities because code will need space, global variables will need space, and this local variables will also need space; so all of them will need space.

Now for the code, so text for a code will consist of text for procedures and programs and the size will be known at compile time. Because when it doing the compilation, so every high level instruction is converted into a set of machine level instructions, so the compiler generate the compiler will know like what is the size of this individual instruction. So, accordingly it can compute what is the size of the code. And as a result, I can know I can allocate this space allow statically before the execution starts. So, this memory can be allocated for this code part.

Now, for the data object parts. So, global variables are constants. So, here also space is known at compile time, because there is only one copy of those global variables. So, they can be done at compile time; however, local variables again local variables also the space is known at compile time. So, how much space is needed that is known. And there can be some dynamically created variables because, so most of the programming languages, so they will allow some dynamic memory allocation, dynamic memory policy for example, this C programs they have got this function malloc.

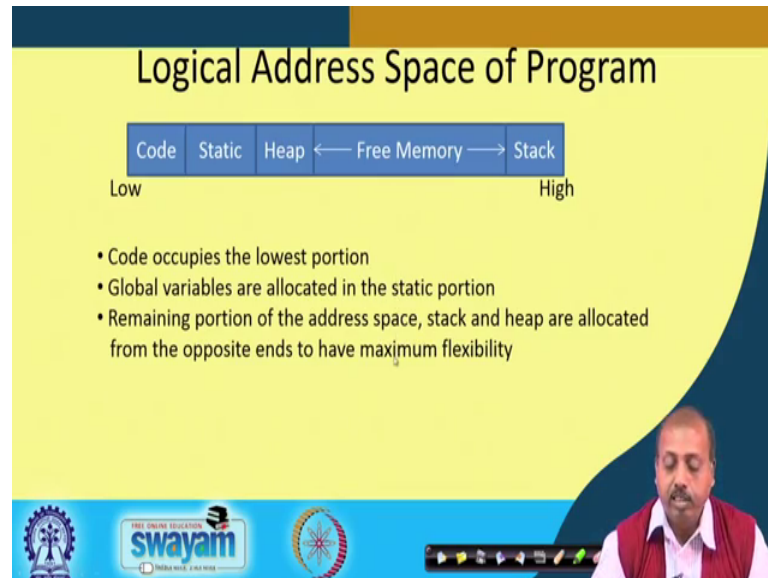
So, you can write like, so if I have got an integer pointer P. So, I can write like malloc something like this. Now this P, so the this space that you get is it was not there with the program at the beginning right. This variable P was there with the program at the beginning, so the compiler could allocate space for that, but for this dynamically allocated memory the space was not available.

So, this space is in response to memory allocation request. So, that is called for that is therefore, the dynamically created variable. So, that is the space is called heap space that is there. And there will be a stack to keep track of the procedure your activation, so which procedure is called and all where from we are returning. So, this return address, return value they are to be stored; so, they will be kept in the stack.

So, this memory needed for this entries like, we need some code space, we need some space for this global variables, local variables and these dynamically created variables.

And also to keep track of the procedure activation, so we need to keep some stack. Now, how are you going to do that? So, that will be the challenge.

(Refer Slide Time: 22:38)



So, we do it like this. So, this code is so, if you take this as the memory space that we have allocated for a particular program. The lower end of the space is allocated to the code part and the code size is known; so it will go up to some point. Then for the static variable or global variable, so the size is again known. So, they will be coming under the static part; so this they will come.

And then the remaining part of the memory, so they are allocated simultaneously to heap and stack, so this heap will grow from the low side and the stack will grow from the high side. So, heap will go from low to high, stack will grow from high to low because if a program has got more of dynamic memory allocations, so it will need more of heap space. On the other hand, if a program needs more of procedure calls, if there are a lot of nested procedure calls, then this stack will be a stack requirement will be high. Because for every such call you will see that some space is needed in the stack. So, since we cannot predict like how the behaviour of a program.

So, whether it is going to take more of dynamic space or more of procedure calls. So, we divide it we allocate the same space for the heap and stack. So, code occupies the lowest portion then the global variables are allocated in the static portion and the remaining portion of the address space that is in the stack and heap they will be allocated from the

opposite ends to have the maximum flexibility. So, it can very well be done that you allocate some space for heap and the remaining space for stack, but the flexibility will be less. So, to maximize the flexibility, so we do it like this.

(Refer Slide Time: 24:24)

Activation Record

- Storage space needed for variables associated with each activation of a procedure – *activation record or frame*
- Typical activation record contains
 - Parameters passed to the procedure
 - Bookkeeping information, including return values
 - Space for local variables
 - Space for compiler generated local variables to hold sub-expression values

Handwritten notes on the slide include:

```
→ x = proc(a, b, c)
  procedure proc(m1, m2, m3)
    int p1, p2;
    return m1 + m2;
  }
x = y + 2 * w
  ↓
t1 = 2 * w
t2 = y + t1
x = t2
```

Now, what is an activation record? So, activation record is the storage space needed for variables associated with each activation of a procedure. So, there that will be called an activation record or frame; so this is some storage space.

For every call that we make to a procedure, so some of the some space will be needed for creating the parameters that you are passing, creating the local variables and all, so all these information. So, this is actually some extra that you that are coming into picture; so that will be required. So, that will be done that will be made in the activation record. And this activation record itself may be created at different places. So, it may be created statically, it may be created dynamically, it may be on heap, may be on stack like that. But they will be they will be required for each such call they will be required.

So, typical activation record contain are like this. So, parameters passed to the procedure then the bookkeeping information keep including return values, space for local variables, and space for compiler generated local variables to hold sub expression values etcetera.

So, it is like this, that suppose I have got a procedure call at this point. So, this I write like x equal to some procedure proc 1 is called, it has got some parameters a b c passed

here. And this procedure proc 1, so it has got say it has gotten this corresponding arguments m_1 , m_2 , m_3 and it has got some local variables. So, p q r are the local variables.

Then what happens is that, when you are making this call somehow I need to remember that what are the values I am passing this a b c what are the values I am passing, so that has to be remembered. Then this return address has to be remembered. Like after finishing this procedure 1 where are you going to come back, that return address has to be remembered. And within once you are in the inside the procedure then you need to have some space for this p q r this local variables and it may there may be some return value of from the procedure.

It may be returning the value of p plus q from the procedure, so the return value has to be there should be some space for that. So, this will happen at runtime, but the compiler needs to allocate space, so for this thing to occur ok. So, it needs to have space, so that while the code will be executing it will be keeping all those information in those spaces, so that the program will execute properly.

And not only this, so like when it is doing the translation of these individual statements that we have in this procedure of proc 1, so it may generate some local variables. Some, some some expression can generate some local variables. As I was telling, like if I if there is a statement like $y \times z$ equal to y plus z into w , so it is for this the code that is generated has got something like this t_1 equal to z into w , t_2 equal to y plus t_1 and then x equal to t_2 .

So, this t_1 and t_2 , so these two are local variables, so that are the call that they are generated by the compilers; they are not there in the original statement, but they are generated by the compiler in the code translation.

So, they are this compile this temporaries they are also to be stored in the activation record. So, all these are the extra information that we need to store while a procedure call is made and they will be made part of the activation record.