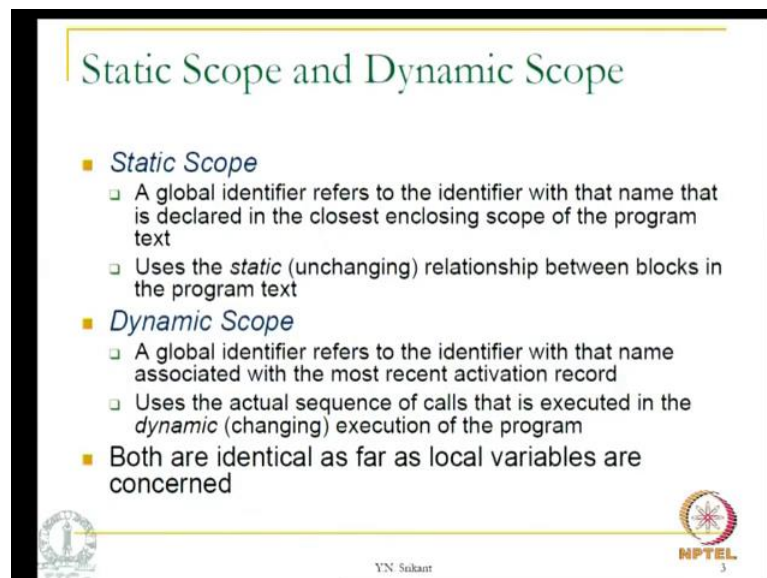


Principles of Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 22
Run-time Environments Part - 3

Welcome to part three of the lecture on runtime environments. Today we will continue with our discussion on static scope, dynamic scope their implementation passing of functions as parameters, etcetera.


(Refer Slide Time: 00:33)



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 with that name associated with the most recent activation record
 - Uses the actual sequence of calls that is executed in the *dynamic* (changing) execution of the program
- Both are identical as far as local variables are concerned

Y.N. Srikant


NPTEL
3

To do a bit of recap on static and dynamic scope.

(Refer Slide Time: 00:40)

Static Scope and Dynamic Scope :
An Example

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


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

x	1	outer block
y	0	

y	3	f(3)
x	4	

z	12	g(12)
---	----	-------

Stack of activation records
after the call to g



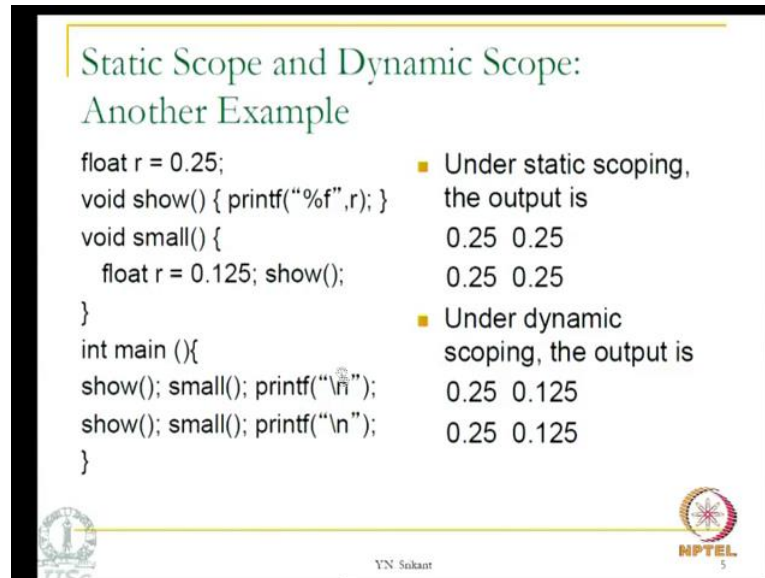
YN Sukant

So, static scope let me go to the example directly; static scope implies that we consider the global variable as you know dictated by the program text. For example, in this program we have X equal to 1, Y equal to 0 here as global variables. And there are occurrences of X here, there are occurrences of here X here. So, when we really call f (3) the control comes to this function and the activation records would have been set up in this fashion. So, this is the main program activation record, then this is the activation record for the invocation of function f. So, within this there is a local variable X. So, there is no ambiguity; this X and this X both refer to the local variable X and it calls g. So, when you go to g there is no X here; so the global variable X is used as the variable that of concern when we consider static scope. So, the value of the global variable is 1 and that is what is used in the a value to be return by g.

Dynamic scope says you know do not bother about the textual position of the variables just consider the activation record; the most recent activation record in which the name occurs. So, in the same example we have f you know Y equal to f (3). So, the control comes here and within f we refer to the local variable X. So, there is no change as far as the local variable is concerned; and then it calls g within g we have this occurrence of X. So, as per the rule in dynamic scoping we consider the activation record for g there is it not any instance of X here; we go to the previous activation record that is the this f and here there is an occurrence of X.

So, even though X is a local variable within f. According to the rules of dynamic scoping this is the variable which is used as the value variable under consideration. So, X plus Z will be return with different value here. So, this is dynamic scoping.

(Refer Slide Time: 03:08)



**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");
}
```

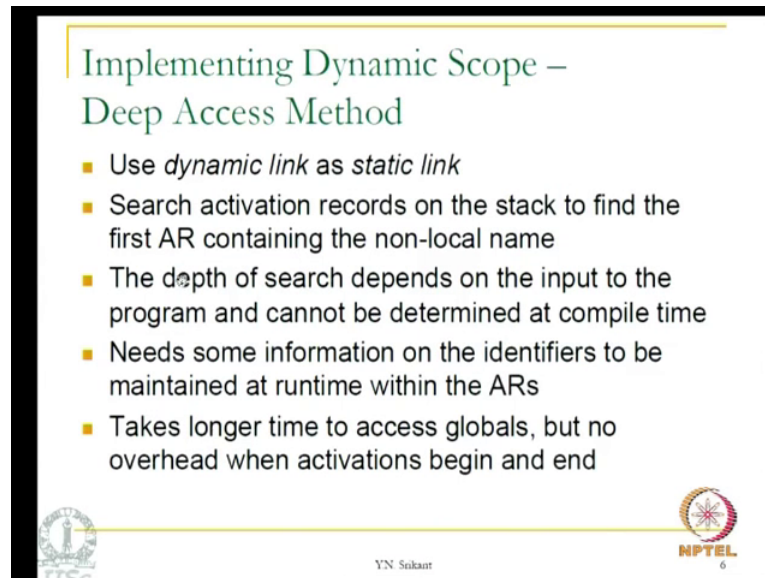
- Under static scoping, the output is
0.25 0.25
0.25 0.25
- Under dynamic scoping, the output is
0.25 0.125
0.25 0.125

YN Sankant

NPTEL 5

There is another example here as well; here is a global variable r. So, small has a local variable r whereas show does not have any local variable declaration. So, when we print this r; is it this r or is it this r that is the question? So, depending on the scoping rules in static scoping it is always this r which is printed from show; whereas under dynamic scoping since show is called from within small it is this r which will be used; just like in the previous example.

(Refer Slide Time: 03:48)



Implementing Dynamic Scope – Deep Access Method

- Use *dynamic link* as *static link*
- Search activation records on the stack to find the first AR containing the non-local name
- The depth of search depends on the input to the program and cannot be determined at compile time
- Needs some information on the identifiers to be maintained at runtime within the ARs
- Takes longer time to access globals, but no overhead when activations begin and end

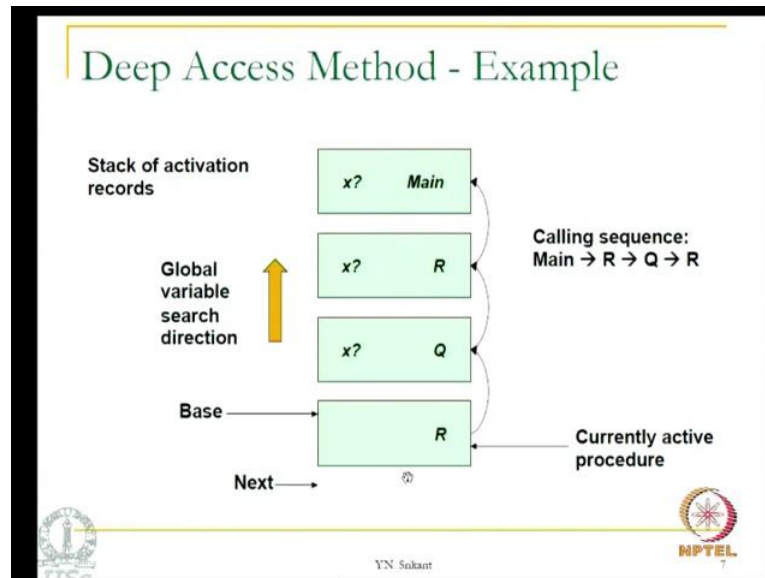
YN Sankant

NPTEL 6

Let us now consider the implementation of dynamic scope. Static scope implementation we discussed it in great detail in the previous lectures. So, using you know the dynamic link and the static link. So, here for implementing dynamic scope we do not require any static link at all; dynamic link itself works as static link. The reason is there is no sanctity as far as the scope rules you know in dynamic scoping; the most recent activation record which contains the instance an instance of the variable, that is that we are accessing right now is all that matters. So, activation records are searched on the stack to find the first activation record containing the non local name of our interest. The depth of such a search cannot be fix at compile time; it really depends on the input to the program. And of course we need to maintain some information on the identifiers within the activation record; I will show you within a minute why this will be required?

And, obviously since we are going to search the stack of activation records; the time required to access global variables is much more than the time required to access local variables. But there is absolutely you know overhead as far as the activation begin and end is concerned. So, let us look at this example right.

(Refer Slide Time: 05:32)



So, here is main main calls R, R calls Q and Q again calls R in a recursive fashion. And within R there is no you know we are considering the variable X; there is no declaration of X here. So, what we really do is start from this activation record and search this stack of activation records to check whether there is instance of X declared or not.

So, Q if Q has the variable X declared in it then this is activation record in which X will be considered. Of course R cannot have X because otherwise this instances would have had X on its own. But we need to continue searching for this activation record if necessary to the main program as well; and if it is found here then that is the X that we require. So, the sequence of calls really matters depending on the length of the sequence of calls; the time required to search for this global variable will also change. And that is the reason why the time to access a global variable will also depend on the sequence of calls that we have made. So, this is the deep access method. And the reason why it is called deep access is that it goes deep into the activation record.


Now, coming to the point that I made here in the previous slide some information on the identifiers need to be maintain at run time within that activation record. So, just look at these this particular example; you need to check whether the activation record for Q contains the variable X or activation record for main contains the variable X and so on and so forth. How do we search if we do you know for search information if we do not you know store it already. So, it is not necessary to store the entire character string of the

variable; it is sufficient to you know code it in some way and store that particular coding within the activation record. But however we definitely want that code to be unique for each name. And it should be easy to search as well.

(Refer Slide Time: 07:52)

Implementing Dynamic Scope – Shallow Access Method

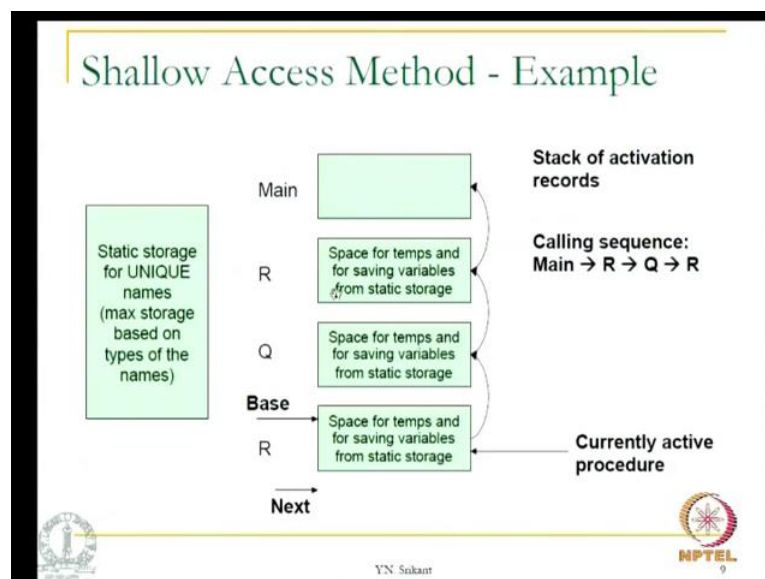
- Allocate maximum static storage needed for *each* name (based on the types)
- When a new AR is created for a procedure *p*, a local name *n* in *p* takes over the static storage allocated to name *n*
 - Global variables are also accessed from the static storage
 - Temporaries are located in the AR
 - Therefore, all variable (not temp) accesses use static addresses
- 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



YN Sankant

There is yet another method for implementing the dynamic scoping; so this is called as shallow access method. So, in let me show you a picture and then come back to this text.

(Refer Slide Time: 08:06)



So, there is a stack of activation records as usual; and we do not require any statics you know link here just the dynamic link will be fine. The activation record does not provide

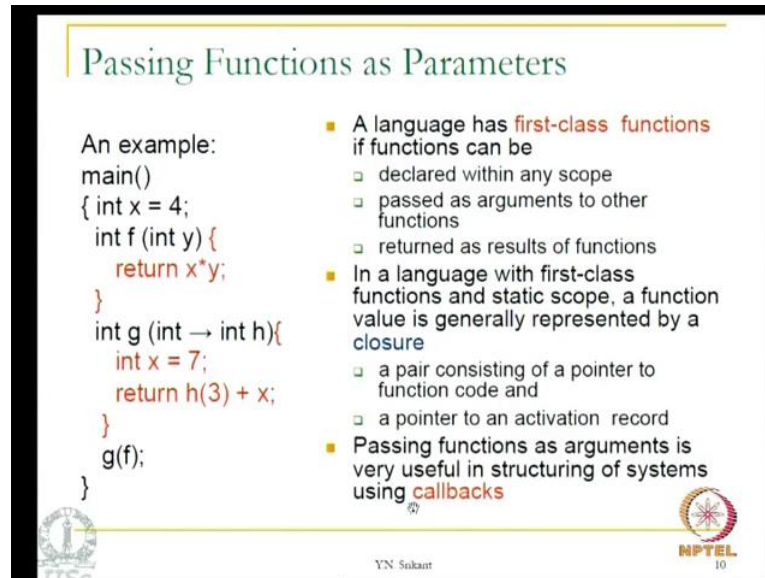
for you know any variable programmer defined variable storage at all; it provide space for temporaries and also some space for saving variables from what is known as the static storage. Then, where are the variables that the programmer has declared in the procedures store; there are actually stored in a unique static you know in a static area; there is exactly one storage you know certain one fixed amount of storage for each unique name. And there may be you know the same name may be declared in various procedures with different types. So, the storage requirement for the name in different procedures may be different.

So, how do we determine the storage that is required for the name; we just take the maximum storage based on the various types for the names. So, this is the storage that we are talking about allocate maximum storage needed for each name so in a static area. Now, when a new activation record is created for a procedure; the local variable n in the procedure p now takes over the static storage allocated to the name n. So, here for example let us say R has the variable X declared. So, the variable X is provided space in these static storage. So, exactly one storage space for each name right but there could have been another X written the same storage right. So, what we really do is this is a local variable. So, we use the we actually store the value of X; that is available here in the storage already in the activation record for this instances of R. And now say that from now on the space for meant for X will belong to the local variable X in the procedure R. Suppose R did not have any variable X but it accessed it; then the storage that was already occupied by some other instance of X is already available here.

So, that is the storage which is accessed; the advantage of this is every name has a unique address; you know and it is static right there is no need to use a stack pointer to access the variable local variables or global variables in this particular scheme. So, all variables but not the temporaries of course; temporaries are allocated space on the activation record itself accesses use static addresses. So, this is a very fast mechanism; of course the previous value of n held in the static storage is saved in the activation record. And it has to be restored when the activation of p ends. So, this store and restore you know from here I need to store something here and then restore it to this later. So, sorry we have to store from something from here to here; and then restore it to back to the original place when the procedure R ends. So, this is an over head in this particular scheme whereas in the other scheme there was no no overhead of this kind.

So, direct and quick access to globals is possible. So, because everything is static addressing scheme but some overhead is incurred when activations begin and ends. So, this is the disadvantage of shallow access method.

(Refer Slide Time: 12:18)



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);
}
```

- 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 and
 - a pointer to an activation record
- Passing functions as arguments is very useful in structuring of systems using **callbacks**

YN Sankant

NPTEL 10

Now, let us continue our discussion on implementation of you know various types of mechanisms in programming languages. The next topic, which is a bit complicated is the parsing of functions as parameters to other functions or procedures; what exactly do we mean? So, let us go through a few points before we discuss this aspect; a programming language is set to have first class functions if we can declare functions within any scope. For example, Pascal allows declaring functions within any other function and so on whereas C does not.

And, then if functions can be passed as arguments to other functions and they can be returned as results of other functions; then it is called as a first class function. So, there is a difference between the function parameter in C and in languages in functional languages and so on and so forth. The problem is in C the static function pointer that is used is only the address of the code; you know that the corresponding to that is associated with that particular function whereas what we really mean in with a by first class function is slightly different it will become clear as we go on. The other programming language which now has some sort of a functions as parameters and so is the C plus plus 11 which is the most recent upgrade to the language C plus plus.

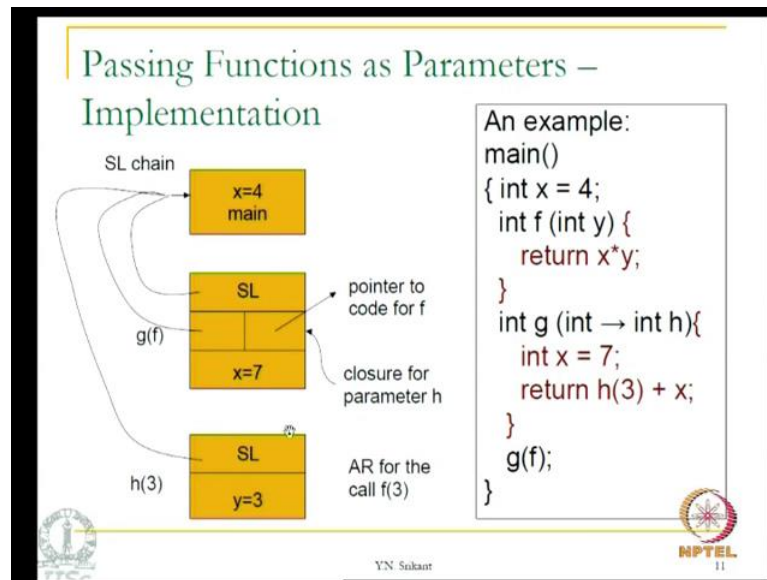
So, I will not deal with the details of such a mechanism C plus plus 11 here. But it suffices to say that the lambda facility are lambda functions in C plus plus 11 are really first class functions. So, in a language with first class functions and let us assume that there is static scope; a function value is generally represented by what is known as a closure. So, what exactly is a closure? A closure is a pair; so it has a pointer to the function code and it also has a pointer to suitable activation record. Why do we have to you know be familiar, why should we be familiar with such facilities? See the such facilities are very useful when we actually use call backs.

And, java does not have such a facility you know; it has a very limited facility for call backs any things like that. But the research in programming languages is actually going to introduce you know such features into the future programming languages C plus plus 11 lambda functions are already there. So, they will be improved and revised in the feature additions of the C plus plus. Let us understand the this particular program this is a you know let assume let there is a static scoping here. So, there is an integer X with initialization as 4 inside the main program we have define another function f which takes one integer parameter and returns X star Y. So, this X is always bound to this X as far as static scoping roles are concerned; there is another function g which is define within name. So, observe that functions being defined in other functions is a necessary feature of first class functions; that is why we are defining the function g inside main; g takes a parameter which is special it takes a function as a parameter.

So, h is a function which is the parameter to this function g; and what is the type of h? It is int to int implying h will take 1 integer parameter and produce a result which is of integer type. So, the body of g says a variable X which is initialize to 7 and then there is a call to h with a parameter 3. So, this is compatible with the declaration here because it takes a value as int as parameter and produces a result int as the output h 3 plus X. The question now is when we try to invoke h; what would be the function code corresponding to it? And number 2 what is the what are the variables which it can access this is the question? Here, is a call to g with f as the function parameter. So, obviously h corresponds to f here when g is called by this main program. So, if we say f (3) it would be implying that we call this function f with 3. So, in this case what is the set of variables that f can use? If you use static scoping then this f can use this X as well and if it is dynamic scoping obviously it would use this X.

Therefore, it is necessary to you know provide when we call f call g with a parameter f; we will also have to say what are the activations records corresponding to the previous activations; in this case just main that are relevant to this particular call. So, those are all associated with f. So, let me show you in example here.

(Refer Slide Time: 18:27)

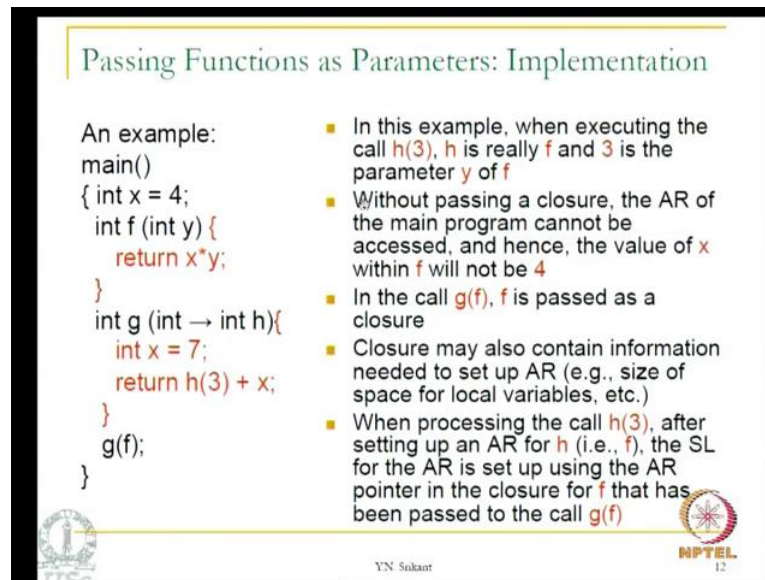


So, to begin with it is main the activation record for main with X equal to 4. Then, we have the activation record for g and within g we have a static link which correctly points to main we have according to the scope rules, because g can access all the variables of main as well. And there is a parameter to g so that is the f part when we actually store the parameter within the activation record of g; there is parameter has 2 parts; one is a pointer to the code for f and the other is a pointer to main.

So, the reason for this is at when we consider f here; the only activation record which is relevant to f is that of a main. Because just for the sake of argument assume that this is a call; if it is a call then you know for this function f the only activation record which is relevant is that of main and the only variable global variable which is that of is X. Therefore, this activation record really points this pointer points to main. So, this is the you know closure for the parameter h and when we invoke h (3) right; we call h with a parameter 3 it is really which is called; we create the you know activation record for f. The only difference is everything else is done exactly the same way as before the static link for f is now copied from the closure of f in the caller.

So, this value is copied here. So, that makes it copy you know point to main; this also takes care of possibilities of you know any recursion and things of that kind. So, this is how parameters are passed as functions are passed as parameters.

(Refer Slide Time: 20:43)



Passing Functions as Parameters: Implementation

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);
}
```

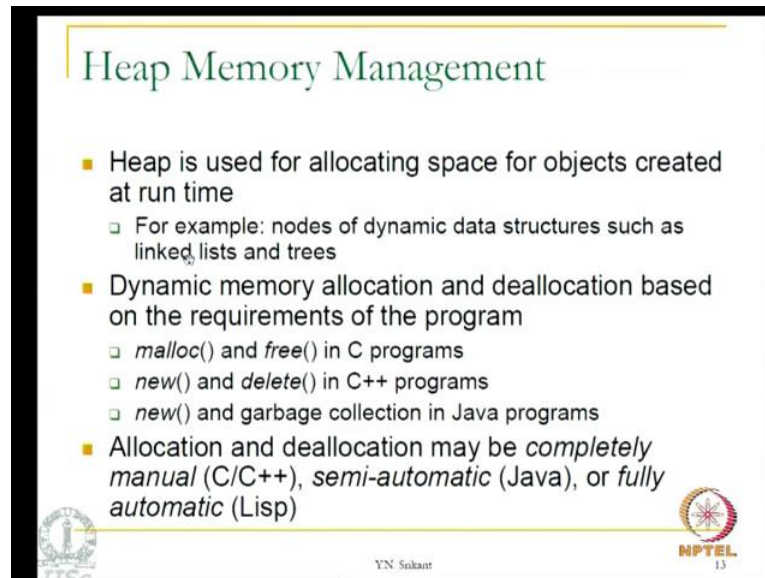
- In this example, when executing the call `h(3)`, `h` is really `f` and `3` is the parameter `y` of `f`
- Without passing a closure, the AR of the main program cannot be accessed, and hence, the value of `x` within `f` will not be `4`
- In the call `g(f)`, `f` is passed as a closure
- Closure may also contain information needed to set up AR (e.g., size of space for local variables, etc.)
- When processing the call `h(3)`, after setting up an AR for `h` (i.e., `f`), 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)`

YN Sankant

NPTEL 12

So, again these to reinforce what I told you now. So, when executing the call `h 3`; `h` is really `f` and `3` is the parameter `Y` without parsing a closure the activation record in the of the main program cannot be accessed. And hence the value of `X` within `f` will not be `4`; in the call `g (f)` `f` is passed as a closure. And of course closure may contain further information needed to set up that activation record such as space for variables and so on and so forth. So, when processing the call `h three` after setting up the activation record for `h` that is `f` the static link for the A R is set up using the A R pointer in the closure of `f` that has been passed to the call `f`. So, this is very important and that make sure that we access the global variables from `f` appropriately.

(Refer Slide Time: 21:41)



The slide is titled "Heap Memory Management" in a green serif font. It contains three main bullet points, each with a yellow square icon. The first bullet point is "Heap is used for allocating space for objects created at run time", with a sub-bullet "For example: nodes of dynamic data structures such as linked lists and trees". The second bullet point is "Dynamic memory allocation and deallocation based on the requirements of the program", with sub-bullets for "malloc() and free() in C programs", "new() and delete() in C++ programs", and "new() and garbage collection in Java programs". The third bullet point is "Allocation and deallocation may be completely manual (C/C++), semi-automatic (Java), or fully automatic (Lisp)". At the bottom left is the IIT Bombay logo, at the bottom center is the name "Y.N. Sankant", and at the bottom right is the NPTEL logo with the number "13" below it.

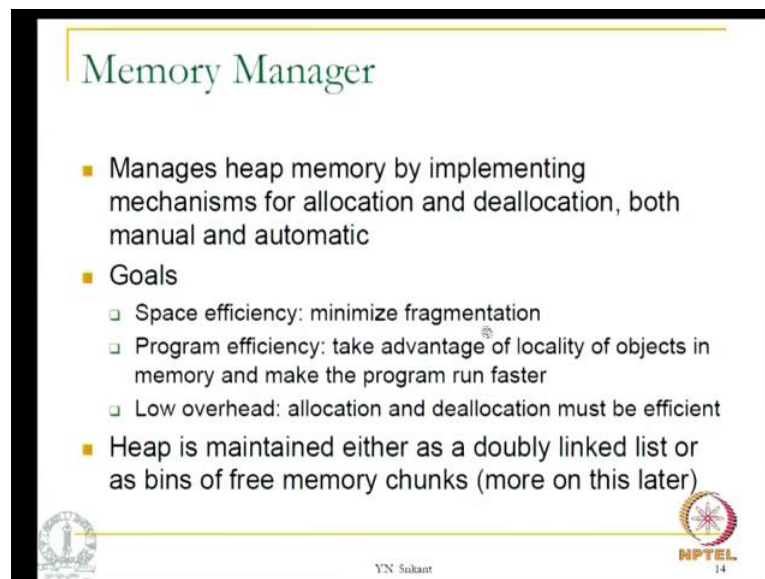
- Heap is used for allocating space for objects created at run time
 - For example: nodes of dynamic data structures such as linked lists and trees
- Dynamic memory allocation and deallocation based on the requirements of the program
 - *malloc()* and *free()* in C programs
 - *new()* and *delete()* 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)

So, now we come to the next topic in run time management; that is the heap memory management. So, what exactly is a heap? Well, we are not looking at the heap sort here you know. So, the heap that we consider here is used for allocating space for objects created at run time. For example, we create link list, we create trees so and any other dynamic data structure of interest to the program. So, nodes you know in such dynamic data structures or all examples of you know the space which is allocated from the heap. So, dynamic memory allocation and deallocation actually is based on the requirements of the program.

So, when we are creating a tree if you want to set up particular node; then we call an appropriate function say malloc or something like that get the storage from the heap. Then, you know put the variable values into the various fields of that block and attach it to the existing tree that completes our you know tree manipulation. So, how do we actually do allocation and deallocation in programs? For example, in C we have malloc and free. So, malloc is used to u know get you a chunk of storage from the heap and free suppose to return the chunk of storage that is of no more use to the heap. Similarly, in C plus plus we have the equivalence new and delete; and in java there is no deletion of storage the new function remains. But the unwanted storage is automatically claimed by the system through a process called garbage collection.

So, we are going to study this all these aspects in this part of the lecture in C and C plus plus allocation deallocation are completely manual; the programmer is responsible for calling malloc, free, new and delete whereas in java new has to be called by the programmer; whereas delete is not necessary you know it is not even provided. So, garbage collection takes care of claiming the free storage. So, we can say it is semi automatic allocation in deallocation in java whereas in the language such as lisp everything is done automatically by the runtime system.

(Refer Slide Time: 24:45)



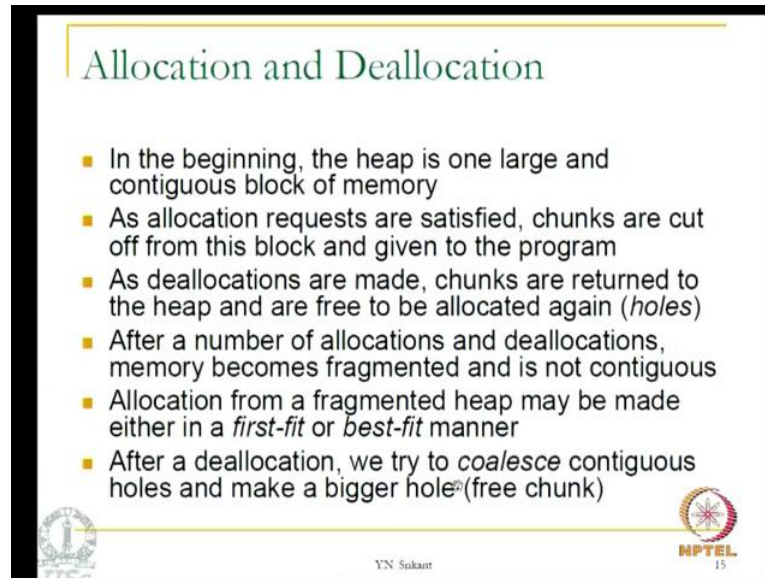
The slide is titled "Memory Manager" in a green serif font. It contains a list of bullet points: a main bullet point with a yellow square icon stating "Manages heap memory by implementing mechanisms for allocation and deallocation, both manual and automatic"; a sub-bullet point with a yellow square icon labeled "Goals" containing three sub-points with grey square icons: "Space efficiency: minimize fragmentation", "Program efficiency: take advantage of locality of objects in memory and make the program run faster", and "Low overhead: allocation and deallocation must be efficient"; and a final bullet point with a yellow square icon stating "Heap is maintained either as a doubly linked list or as bins of free memory chunks (more on this later)". At the bottom left is a circular logo with a book and a lamp. At the bottom center is the text "YN Sankant". At the bottom right is the NPTEL logo with the number "14" below it.

What is a memory manager? Obviously, heap memory has to be managed by some part of our program. So, heap manager manages heap memory by implementing the mechanisms for allocation, de allocation you know both manual.

And, automatic in fact it takes care of even garbage collection any things of that kind; what are the goals of a memory manager? So, space efficiency it must minimize what is known as fragmentation of the heap; as you go on in the next few minutes it will become clear about you know the reason for fragmentation. Fragmentation basically says the heap memory is not one contiguous block of memory. But it is actually pieces of memory put together; then program efficiency programs have to run fast and efficiently. So, the memory manager must take advantage of locality of objects in memory and make the program run faster. And then the memory manager itself should have less overhead otherwise allocation, deallocation become quite efficient.

Usually, heap is maintained as a doubly link list in some types of front end systems; some time it is also maintained as bin of free memory chunks. So, we will discuss this in detail little later.

(Refer Slide Time: 26:22)



The slide is titled "Allocation and Deallocation" in a green serif font. It contains a bulleted list of six points describing heap memory management. The list is as follows:

- In the beginning, the heap is one large and contiguous block of memory
- As allocation requests are satisfied, chunks are cut off from this block and given to the program
- As deallocations are made, chunks are returned to the heap and are free to be allocated again (*holes*)
- After a number of allocations and deallocations, memory becomes fragmented and is not contiguous
- Allocation from a fragmented heap may be made either in a *first-fit* or *best-fit* manner
- After a deallocation, we try to *coalesce* contiguous holes and make a bigger hole (free chunk)

At the bottom of the slide, there are three logos: a circular logo on the left, the text "Y.N. Sankant" in the center, and the NPTEL logo on the right. The NPTEL logo includes the text "NPTEL" and the number "15" below it.

So, let us assume that heap is one large contiguous block of memory to begin with. So, the program has just begin execution it has been given a certain amount of memory as heap and from which allocation and deallocation can be done. So, to begin with heap is one big block; as the allocation request keep coming in chunks are cut out from this block and given to the program. So, far so good the blocks are still a remaining block of memory that is heap is still single contiguous block of memory. But the program may also deallocate; you know it may remake 3 requires and 2 deallocations and then again another three requires and another deallocation etcetera, etcetera.

The free chunks are returned to the heap and they can be reused again. So, these free chunks are called usually as holes; and after number of such allocations and deallocations it is quite understandable that memory becomes fragmented. Because the allocations you know the allocated allocations and then the deallocations they need not be done in the same order. So, whatever is you know useful to the program is released but it is not necessary that we allocate 1, 2, 3 and then 3 blocks in that order. And then deallocate exactly in the reverse order 3, 2, 1 that will never happen. So, it is usually the allocations are in particular sequence but deallocations can be in random sequence. And

that makes the you know heap a fragmented memory. So, after deallocation we try to coalesce contiguous blocks and make a bigger hole or bigger chunk. So, this is the this is what we try to do in order to remedy the situation.

(Refer Slide Time: 28:34)

First-Fit and Best-Fit Allocation Strategies

- The *first-fit* strategy picks the **first** available chunk that satisfies the allocation request
- The *best-fit* strategy searches and picks the smallest (**best**) possible chunk that satisfies the allocation request
- Both of them chop off a block of the required size from the chosen chunk, and return it to the program
- The rest of the chosen chunk remains in the heap

YN Sankant

NPTEL 16

The question is why do we have to do all this you know; the allocation can happen but then we deallocate it is necessary that we have as a big chunk as possible. Because you know smaller chunks will a 3 small chunks will never can never be allocated in the place of one big chunk. So, this is the problem. So, there are 2 strategies which are used; one of them is called the first fit strategy, the other is called as the best fit allocation strategy. The first fit strategy picks the first available chunk that satisfies the allocation request.

So, as the name suggest the doubly link list is travels from one end and the first block on this doubly link list which has a size greater than what the programmer has requested satisfies the programmers request. And therefore the you know the best fit the allocator chops of a block of the required size from the chunk and returns it to the program rest of the chosen chunk of course, remains in the heap itself. But then what is the best fit strategy? Best fit strategy says ok do not pick the first one which satisfies which can satisfy the programmers request. But search the whole list and pick the smallest possible chunk that satisfies the allocation request.

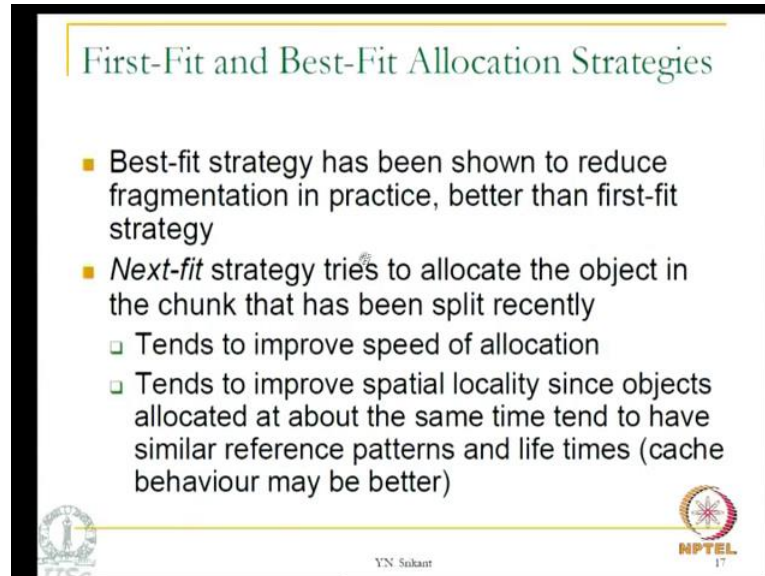
So, the advantage of doing that is the fragmentation is reduced; we do not have to cut small blocks from large blocks and then you know make the larger block itself smaller.

Whereas if we pick the smallest block chunk which satisfy the location request. And then chop off the required size from it we would possibly the reducing the fragmentation in the heap. So, obviously the best fit strategy looks better and in practice it has been shown that it reduces fragmentation better then the first fit strategy.

But of course you can always produce theoretically sequence of allocations and deallocations such that best fit strategy fails but first fit you know succeed and vice versa. So, these are theoretical but the in practice best fit is much better; there is also strategy known as the next fit strategy. So, what is next fit? So, suppose that a chunk has been used for allocations. So, a small piece of it has been cut and returned to the programmer program the next allocation is done from this particular chunk itself. So, it tries to allocate the object in the chunk that has been split recently; what is the advantage of doing it? Obviously it tries to improve the speed of allocation; we are not going to search and then you know allocate as in the case of first or best fit.

And, it also tries to tends to improve the spatial locality because object allocated at about same time tend to have similar reference patterns and life times. So, if the objects you know are actually cut out from the same storage you know heap then cache performance may become better. Because these will be cache together and since there are they may be access together; the cache performance becomes better. And the overall you know the speed of the programming increases; that was about the you know doubly link list approach to storing heaps rather managing heaps.

(Refer Slide Time: 32:42)



First-Fit and Best-Fit Allocation Strategies

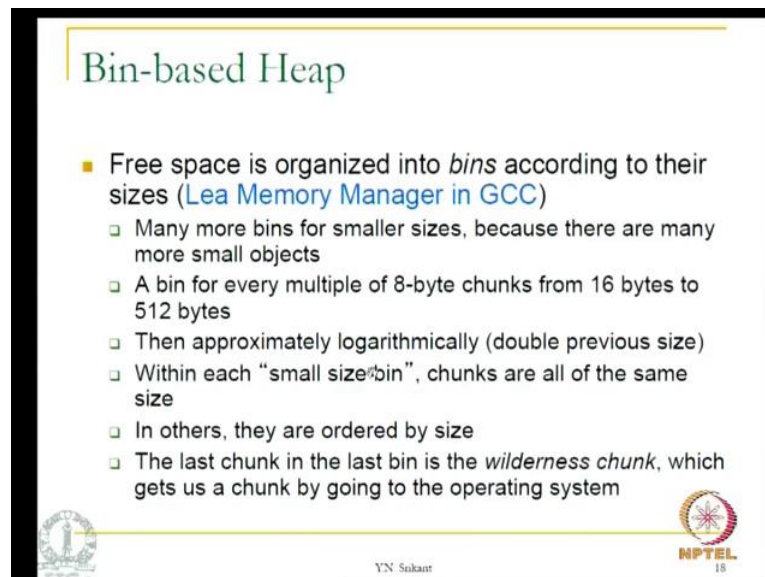
- Best-fit strategy has been shown to reduce fragmentation in practice, better than first-fit strategy
- *Next-fit* strategy tries to allocate the object in the chunk that has been split recently
 - Tends to improve speed of allocation
 - Tends to improve spatial locality since objects allocated at about the same time tend to have similar reference patterns and life times (cache behaviour may be better)

YN Sukant

NPTEL 17

So, there is also another form of storing the heap organizing the heap called as the a bin based heap. So, free space is organized into bins according to their sizes. So, this has been implemented in G C C and it is usually called as the lea memory manager; because lea is a person who invented this particular scheme.

(Refer Slide Time: 33:22)



Bin-based Heap

- Free space is organized into *bins* according to their sizes ([Lea Memory Manager in GCC](#))
 - Many more bins for smaller sizes, because there are many more small objects
 - A bin for every multiple of 8-byte chunks from 16 bytes to 512 bytes
 - Then approximately logarithmically (double previous size)
 - Within each "small size" bin, chunks are all of the same size
 - In others, they are ordered by size
 - The last chunk in the last bin is the *wilderness chunk*, which gets us a chunk by going to the operating system

YN Sukant

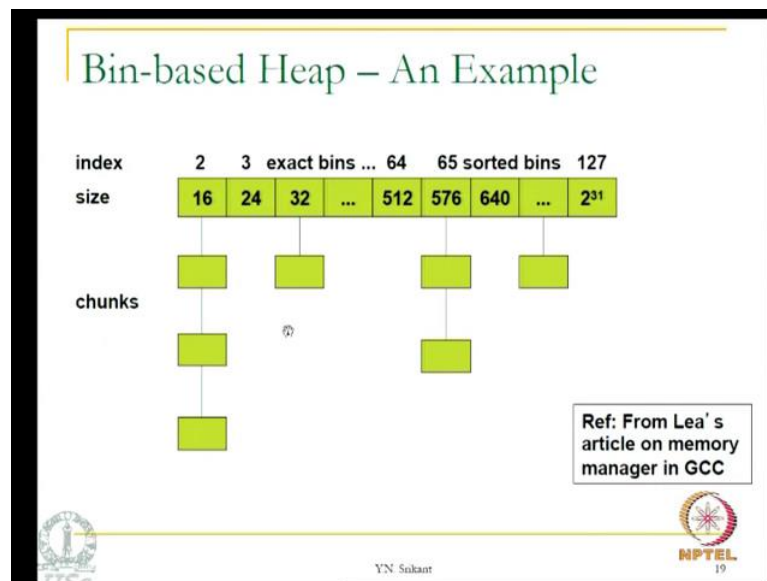
NPTEL 18

So, let me show you the organization of the bin based heap; then go back to the description. So, the this is the index all right so and each one of these indexes stores a pointer to a link list all right; there are number of link list here. So, one part of the bin

based heap stores what is known as exact bins or fixed size bins; whereas the second part of the bin based heap stores variables sized bins; the reason for this is there are there is a requirement for you know small objects, because there are many more small objects than large objects. So, we have many more bins for smaller sizes and few are bins for very large sizes. So, 1 to 64 all of them are fixed size or exact size bins. So, a bin for every multiple of 8 byte chunks from 16 bytes to 512 bytes sizes. So, there is a bin for chunks of 8 byte size. So, all the chunks in these list; so let us say this is a 16 byte size list. So, this is the index for the or a pointer to this link list; so each node on these link list will be of 16 bytes size.

Here, is a list for the 32 byte size. So all nodes on this list will be of size 32. Then, after this a fixed size bins; we have a number of variables size bins and the size of these variable chunks in the variable size bins is approximately double the previous size.

(Refer Slide Time: 35:33)



So, for example you know you can see that we start with 566 slowly increase the size and it goes to 2 to the power 31 here ok; in 1 you know from 65 to 127 we have already risen from 576 to 2 to the power 31. So, the sizes have really increased exponentially; with each small size bin chunks are all of the same size within each small size bin whereas in the others there are ordered by the size. So, here in these there all of the same size and where as from this 576 onwards these sizes are different; but all chunks here will be definitely less than the sizes in the next list. So, in this case this is 576, this is

640. So, all these are less than 640 in this case. So, and this are actually ordered by size here. So, this could be 576; this next one could be 590 and so on and so forth. The last chunk in the last bin is what is known as the wilderness chunks, which gets us a chunk by going to the operating system.

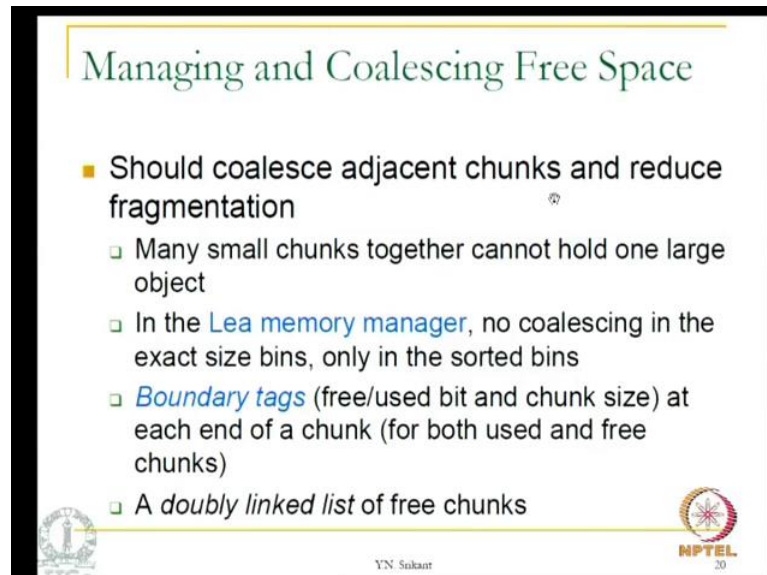
So, let me explain what happened here. So, this is the bin containing you know the wilderness chunk. So, the size is very large 2 to the power 31 ; so extraordinarily large size that does not mean it has a list of such blocks available; this is what is known as a wilderness bin. So, what we really do is allocation happens by looking at the size that we require; if that map to one of the fixed size bins we go there pull off one chunk from that bin and return it to that program. If that bin does not have any then the for example we require let us say 24 ; 24 has a empty list attach to it. So, we go to 32 pull this chunk which is available of size 32 .

But we do not cut anything from it we just return it to the program; and when it comes back in size 32 we attach it to the same list here. So, if none of the exact size bin satisfy the programs request then we have no option but to go for the variable size bins. So, we go to the appropriate variables size bin which satisfies the program request. Then, search you know the bin and the one which best fit in a best fit manner; since it is sorted already the first one itself will be the best fit. So, this is a sorted in increasing size. So, the first one will be the best fit size; we take this cut the required size from it returned it to the program. Now, there is going to be a small piece which is left here; this will be a return to the appropriate bin. So, if it fix into one of this size bins then it is put there otherwise it will be actually put in one of this bin as possible.

And, now the wilderness change. So, let us say we come up to this point we did not find any of this bins you know having chunks ok. So, all of them are empty; so in other words the programmers really used up the entire heap. So, we hit the wilderness bin; so which implies when we hit the wilderness bin the implication is that there is no more memory available within the program. So, we make a then an automatic call is made to the operating system to release more memory to the heap of this particular program; the once that arrives a chunks is cut off from that. And the request of whatever size is made to the program to the operating system; the operating system will return chunk of the required size of slightly larger and that will be returned to the program.

So, this is known as the call to the operating system, which returns the chunk of the required size.

(Refer Slide Time: 40:09)



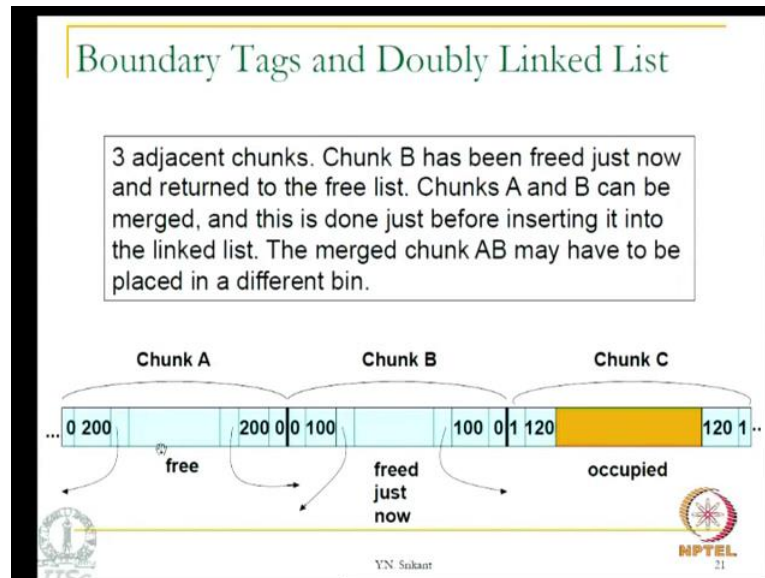
The slide is titled "Managing and Coalescing Free Space" in a green serif font. It contains a bulleted list of four items. The first item is a yellow square bullet followed by "Should coalesce adjacent chunks and reduce fragmentation". The second item is a white square bullet followed by "Many small chunks together cannot hold one large object". The third item is a white square bullet followed by "In the *Lea memory manager*, no coalescing in the exact size bins, only in the sorted bins". The fourth item is a white square bullet followed by "*Boundary tags* (free/used bit and chunk size) at each end of a chunk (for both used and free chunks)". The fifth item is a white square bullet followed by "A *doubly linked list* of free chunks". At the bottom left is a small circular logo with a building. At the bottom center is the text "Y.N. Sankar". At the bottom right is the NPTEL logo, which is a circular emblem with a star and the text "NPTEL" below it.

- Should coalesce adjacent chunks and reduce fragmentation
- Many small chunks together cannot hold one large object
- In the *Lea memory manager*, no coalescing in the exact size bins, only in the sorted bins
- *Boundary tags* (free/used bit and chunk size) at each end of a chunk (for both used and free chunks)
- A *doubly linked list* of free chunks

So, that is about the lea memory manager. Now, how do we manage you know the and coalesce free space? So, as I said we should really combine adjacent chunks and try to reduce the fragmentation. Because many small chunks together cannot hold one large object you know they are all different chunks they are not contiguous. And a program requires the object memory to be allocated in one continuous or contiguous memory space; in the lea memory manager there is no coalescing in the exact size bins. But only in the sorted bins. And how do we do such coalescing? They require extra information in the form of what are known as boundary tags.

And, the which implies free and used bit and the chunk size I am going to show you an example which uses boundary tags; at the end of each chunk. And this is used for both free and used chunks we maintain the link list in a doubly linked fashion.

(Refer Slide Time: 41:23)



So, here is an example chunk A is still in use chunk B has been released just now. And chunk A is free chunk B has been freed just now and chunk C is still occupied; what is shown here is the memory maps starting from 1 and 2 the other. In other words the address of chunk B starts just after the address of chunk A. So, these 2 are actually contiguous since B has been released just now; it is possible to combine these 2 chunks and make a bigger chunk.

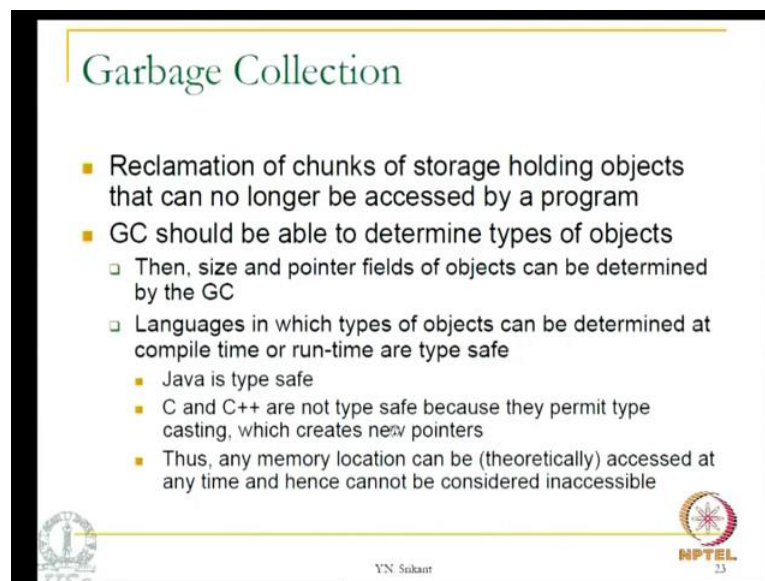
So, to do that what is it that we require? We require the information about the usage of the chunk. So, 0 indicates it is a used chunk so we maintain a 0 at both ends of the chunk; next we maintain the size of the chunk. So, this is 200 so 200 is maintained at both ends side of the chunk. Similarly, this says this is 0, and this is 100. So, 100 and 0; this 1 indicates it is an occupied chunk the size is 120; why should we maintain information about the chunk size and usage bit at both ends of the chunk? Well, you know we are manipulating a doubly link list. For example, this goes to the previous free node and this goes to the next free node. Now, this has been released so this is yet to be filled these 2 are yet to be filled. And suppose these were not contiguous as I have shown here you know they are not adjacent let us say; actually this would have pointed to this ok.

And, this would be pointing to the next node in the link list. Now, these 2 are adjacent so there is no need to make 2 nodes out of it; we can combine these 2 make a single node of

size 300 actually in practice little more than 300. Because the storage require used for this 200 00 100 will also be available to the big chunk itself. So, we are going to modify these 2 this part of it will be merged. So, there is no link here there is nothing here as well this; and this will be entire thing will be 1 chunk, this will be 0 the size now will be 300 and this will be 0 and this will also be 300. So, this points to the previous chunk and whatever this was pointing to ok; this is the next free chunk right that will be point it to by this link now. So, this will point to the free chunk which this is pointing to.

So, that makes this entire thing 1 big chunk of 300 size and it would be link it to the doubly link list in the appropriate manner. So, this is how the merging takes place; there is a comment here the merged chunk A B may have to be placed in a different bin; if you are using a lea memory manager. Because this has now you know increase to size 300. So, it must be put in possibly different bin; so if you look at this so each of these bins are different size. So, if the merger happens here and the size goes behind 576; it may have to be put into one of these bins as necessary.

(Refer Slide Time: 45:02)



Garbage Collection

- Reclamation of chunks of storage holding objects that can no longer be accessed by a program
- GC should be able to determine types of objects
 - Then, size and pointer fields of objects can be determined by the GC
 - Languages in which types of objects can be determined at compile time or run-time are type safe
 - Java is type safe
 - C and C++ are not type safe because they permit type casting, which creates new pointers
 - Thus, any memory location can be (theoretically) accessed at any time and hence cannot be considered inaccessible

Y.N. Sankant

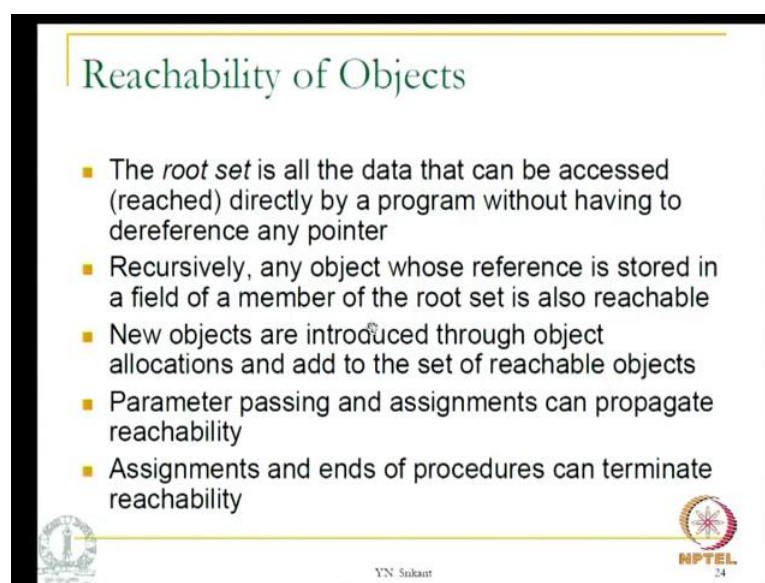
NPTEL 23

So, this is about manual allocation and deallocation process; what are the problems with such manual strategies? It is possible that they memory leaks which occur cannot be detected. So, what is a memory leak? Failing to delete data that cannot be referenced. So, we have generated garbage which cannot be referenced any more but we for what to delete it. So, the this memory leak implies this data you know these nodes cannot be used

again they are not to be return to the heap. So, they cannot be reused but they just occupy memory and waste; it is actually they waste the memory space available to the used by the program. And this becomes very important in learning nonstop programs or program which run for a very long time. The reason is suppose a small amount of memory is leaked in a loop right. So, in every iteration of the loop if there are millions and millions of iteration a little bit of memory is not freed; it becomes garbage.

And, in after completing the entire set of million or more runs; it is possible that the unused you know the data space which was leaked becomes quite large. And the program runs out of memory even though the unused you know the memory is within the program; and has not be return to the heap. The second problem with manual allocation is it is almost impossible to detect this dangling pointer dereferencing. So, what we have done is in this case we deleted the we did not delete the data; but we simply you know threw away the a node and never bother to return it to the heap; whereas here we have deleted the data we have return the you know node to the heap. But some other part of the program is still referencing this deleted data area; this is known as a dangling pointer dereferencing. And in such a case since referencing deleted data is illegal and whatever we get there may be illegal data the program may crash. Both these problems are very serious and very hard to debug; a solution rather parser solution is automatic garbage collection.


(Refer Slide Time: 47:58)



Reachability of Objects

- The *root set* is all the data that can be accessed (reached) directly by a program without having to dereference any pointer
- Recursively, any object whose reference is stored in a field of a member of the root set is also reachable
- New objects are introduced through object allocations and add to the set of reachable objects
- Parameter passing and assignments can propagate reachability
- Assignments and ends of procedures can terminate reachability

YN Sankant

 NPTEL
24

So, what exactly is garbage collection? Garbage collection is reclamation of chunks of storage; holding objects that can no longer be accessed by the program. So, there is no deletion parsers; for example in java we only allocate but we do not deallocate. But definitely after some time we program has no use for certain of certain number of the nodes; and these objects you know will have to be collected by the so called garbage collector and returned to the heap. So, the garbage collector should be able to you know determine the types of objects; the reason being if it does not know the type then size of the objects cannot be determined.

So, then the size and pointer fields of the objects can be determined by the G C; why should it determine the pointer fields of the objects, why not just the size? The problem is when an object is of no use anymore; all the data which is pointed to by the pointer fields within the object will also become of no use you know they is also be useless. So, it is necessary to know which are the pointer fields within the object. So, all this can be done if the type of object is known to us; you know we will know which field is int which field is pointer etcetera, etcetera. Languages in which types of objects can be determined at compile time or run time are set to be type safe. So, it is possible that in some cases we cannot determine the type of the object that compile time; it can only we done at run time.

For example, in the case of java; java is type safe when we try to make a virtual method call the type of object cannot be determined at compile time; it will have to be determined only at run time. And based on the type of that object the appropriate method will have to be called. So, this is a case where the type of the object is determined at run time; C and C plus plus are not at all type safe they permit type casting. So, we can create new pointers. So, now suddenly what was looking as a floating point number can suddenly become character or vice versa. So, any memory location theoretically can be accessed at any time; because a if floating point numbers suddenly becomes a pointer we do not know what it is going to point to. So, if there is a error in the program then you know the program may crash. So, it and therefore any part of the program can be accessed by these pointers.

In fact in some cases; there are examples which we can create to show that even the program code itself can be kind of over written by such new pointers. Then, having said that we require you know garbage collection and we require type information and so on.

let us see what exactly is done at garbage collection time? There is the concept of what is known as a root set; the root set is all the data that can be accessed or reached directly by program without having to dereference pointer. So, in other words let us say you know we just take C for that matter even though it does not have automatic garbage collection; we declare many pointers in C right.

So, the names of these pointers are at the first level and they are declare by the programmer himself or herself; we can say that the set of all such you know programmer defined pointer names is the root set in the case of A C program, because this is the data that can be accessed directly by a program without having to go through or dereferencing any other pointer. So, we are not really taken information by going to the address which is pointer point to. And then taking the address from there and traversing further we simply said ok we have a pointer. And the pointer information is all that we are looking at. Now, we can do this recursively any object whose reference is stored in a field of a member of the root set is also reachable. So, the root set you know all the variable the pointer variables are reachable; that is the root set.

Now, take the objects that are reachable from the pointer variables. So, within that there are many other you know these are all reachable and the within these objects there will be other pointers. So, from the second step now says take the object, take the members of that particular objects and trace further. So, there will also become reachable; new objects are introduced through object allocations and added to the set of reachable objects parameter. So, when we create a new object using malloc or new etcetera, etcetera. So, these you know objects or created new and there are added to the set of reachable objects. Because first time we create an object it is reachable through the pointer, which points to it; parameter passing and assignments can propagate reachability.

So, in other words when I pass a parameter I say now the function which gets it parameter will have let us say call by value parameter. So, I pass a parameter to the function which I am calling; and from that function I am going to create a new variable local variable. And that would be actually pointing to the object that I passed right. So, parameter passing will actually propagate the reachability and of course assignments will propagate reachability. Because if I say A equal to B whatever B points to will also be

point to 2 by A now. Now, reachability actually propagate; assignments and ends of procedures can terminate reachability as well I said here assignment can propagate.

But assignments can also terminate reachability because the same case of A equal to B; A was pointing to something. Now, after the assignment A equal to B; A will now point to what B was pointing to. So, whatever A was pointing to has been kind of terminated. So, there it is not accessible any more through the pointer in A. And once a procedure ends all the local variables which were created within the procedure also die including local pointers. So, all these variables you know the point to kind of truncate the or terminate the reachability of objects that they point to. So, we will stop the lecture at this point; and continue with the methods of garbage collection in the next part of the lecture.

Thank you.