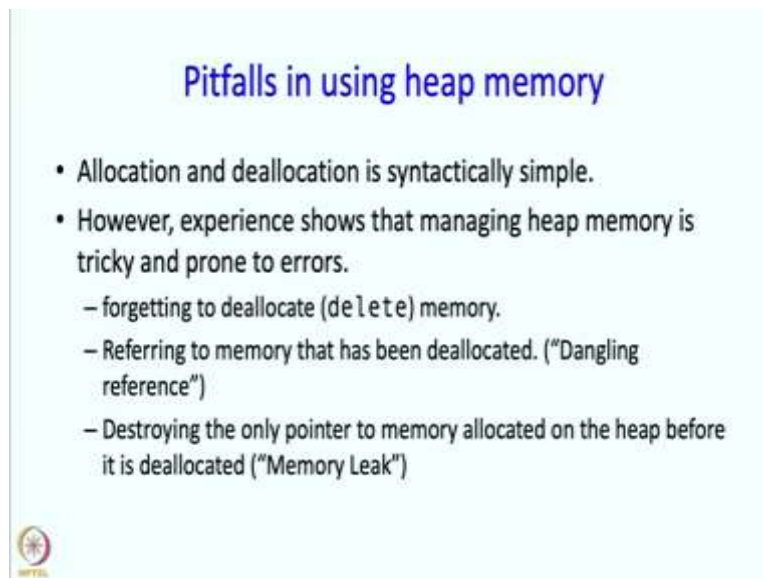


An Introduction to Programming Through C++
Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture 22 Part-3
Representing variable length entities
Pitfalls of using heap memory

Welcome back. In the last segment we discussed the basic primitives for dealing with the heap memory and we use them to solve a particular problem. Now, we are going to discuss the issues in using the heap memory and what kinds of pitfalls there are. So, on the face of it, it seems, and I guess it is true that the allocation and deallocation is syntactically quite simple.

(Refer Slide Time: 00:44)



Pitfalls in using heap memory

- Allocation and deallocation is syntactically simple.
- However, experience shows that managing heap memory is tricky and prone to errors.
 - forgetting to deallocate (delete) memory.
 - Referring to memory that has been deallocated. ("Dangling reference")
 - Destroying the only pointer to memory allocated on the heap before it is deallocated ("Memory Leak")

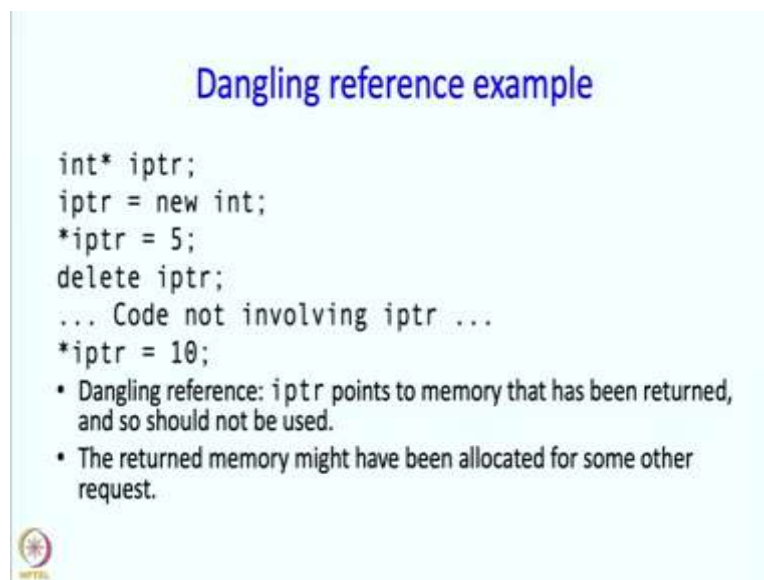
There are sort of simple commands which allow you to allocate and deallocate. Experience however shows that managing heap memory is tricky and lots of errors are made in the management of heap memory. So, these range from some very obvious ones to less obvious ones say for example, you may forget to deallocate or delete memory. What does that do?

So, you allocate a lot of memory, you do not delete it and then if you yourself want more memory then there is nothing left in the heap, so although you have some memory which you are not using and which have not realized that you are not using since you did not return it to the heap, the heap is unable to give you memory. So, that is sort of a basic problem, so as soon as you are done with the memory that you need it you should return it back so that you can get it later on when you actually wanted.

Another problem is the so called Dangling reference problem. This happens if you refer to memory that has been deallocated. So, you got some memory, you are using it, you returned it and you are still referring to it.

So, because you returned it, it might have been given to some other request and therefore you might be destroying the data which or it might be reading data which was not really data that you thought was present in that location and you can also do something called a Memory Leak, which is that you destroy the only pointer to the memory allocated on the heap before it is deallocated, before it is explicitly deallocated. So, we will talk about all these three in some details.

(Refer Slide Time: 02:47)



The slide is titled "Dangling reference example" in blue text. It contains the following C++ code snippet:

```
int* iptr;
iptr = new int;
*iptr = 5;
delete iptr;
... Code not involving iptr ...
*iptr = 10;
```

Below the code, there are two bullet points:

- Dangling reference: `iptr` points to memory that has been returned, and so should not be used.
- The returned memory might have been allocated for some other request.

In the bottom left corner of the slide, there is a small circular logo with the letters "WIT" inside.

So, let us take an example, suppose I have `int *iptr`. So, `iptr` is a variable which contains which is expected to contain pointers to integers. So, I make it point to a new integer, I make it point to a space allocated on the heap. I store 5 into that space then I delete that. Then I write some code which could be a long piece of code but let say it does not involve `iptr` and after that I again store 10 into that address by dereferencing `iptr`.

Now, this is not correct, so you did have `iptr` did have a memory location which was given from the heap but now so it was given here but you returned it over here and you did not acquire it again and place it into `iptr`. So, therefore the location that you are pointing to is not really what you think it is, it is not your location and so you should really not be using it.


So, this reference `iptr` points to memory that has been written and it should not be used so therefore it is called the dangling reference and it is wrong to dereference a dangling reference as has happened over here.

(Refer Slide Time: 04:23)

Memory Leak 1

```
int *iptr;  
iptr = new int; // statement 1  
iptr = new int; // statement 2
```

- Memory is allocated in statement 1, and its address, say A, is stored in `iptr`. However, this address is overwritten in statement 2.
- Memory allocated at address A cannot be used by the program because we have destroyed the address.
- However, we did not return (`delete`) that memory before destroying the address. So the heap allocation functions think that it has been given to us.
- The memory at address A has become useless! "Leaked"



So, let us talk about memory leaks. So, again we have this `iptr` which is a variable for storing addresses of integers, so we acquire memory and get its pointer into, get its address into `iptr` and immediately we acquire more memory and get that address also into `iptr`. Now, clearly this is wrong. So, the memory allocated in statement 1, its address was stored say let us call it A was stored in `iptr` but in statement 2, this A got overwritten.


So, now the memory allocated address A cannot be used by the program because we do not have any address, we do not have its address. On the other hand, we did not delete that memory before destroying the address and so the heap allocation functions, the heap manager thinks that that address has been given to us. So, now that piece of memory is has become useless because it does not, we cannot use it and the heap manager cannot use it. So, such memory is set to have leaked out of our system and obviously you do not want such leaks.

(Refer Slide Time: 05:49)

Memory Leak 2

```
{int *iptr;  
  iptr = new int; // statement 1  
}
```

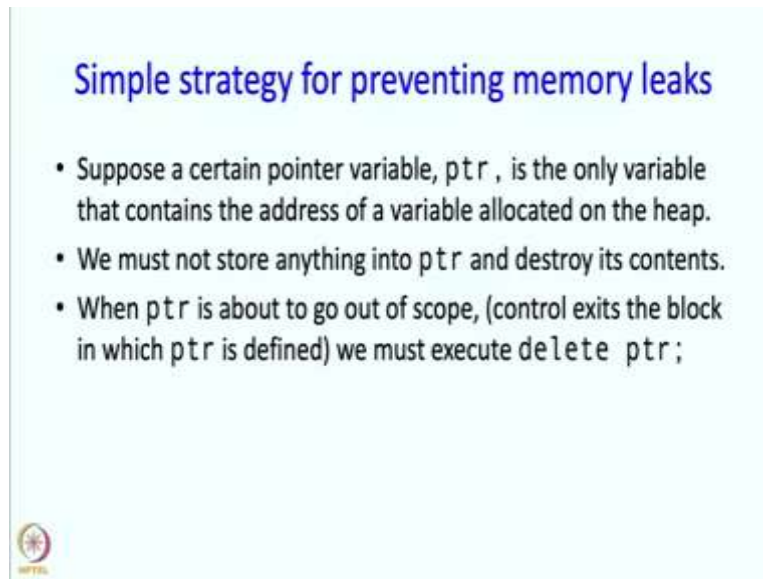
- Memory is allocated in statement 1, and its address, say A, is stored in iptr.
- When control exits the block, then iptr is destroyed.
- Memory allocated in statement 1 cannot be used by the program because we do not know the address any longer.
- However, we did not return (delete) that memory before destroying the address. So the heap allocation functions think that it has been given to us.
- So the memory at address A has become unusable!



Here is another way a leak can happen. So, I have a block inside which I have a variable declared iptr. It is again of type int *. So, again I allocate an integer and get its address into iptr but say the block ends exactly at this point. What happens now? So, when the control exits from the block all the variables which you were declared in the block are destroyed.


So, after the control executes this and comes out, there is no iptr, so that means the contents of the iptr are gone. So, I had written down the address on some piece of paper and the paper is no longer there so I do not know that address, that is exactly what has happened. So, iptr is a destroyed and so nobody can use the memory allocated in statement 1, so we cannot use it because we do not know the address and the heap manager thinks that it is given to us so therefore the heap manager cannot use it either. So, the memory at address A has become unusable.

(Refer Slide Time: 07:14)



Simple strategy for preventing memory leaks

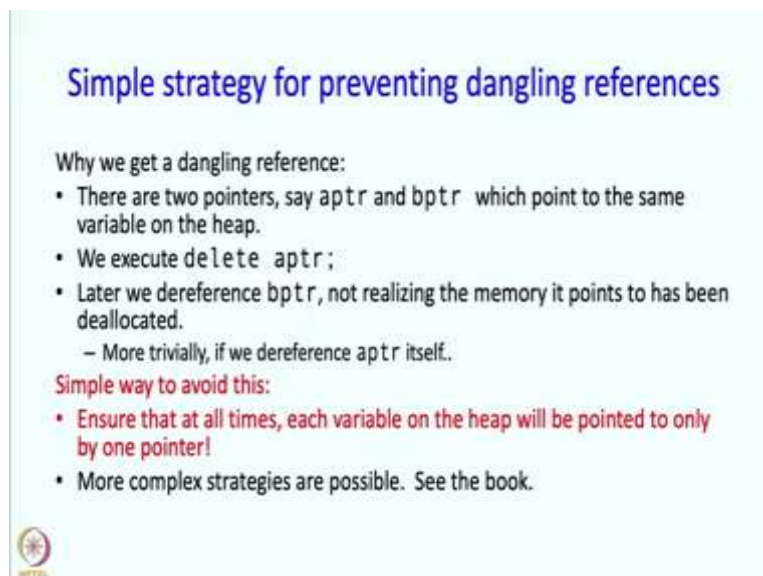
- Suppose a certain pointer variable, `ptr`, is the only variable that contains the address of a variable allocated on the heap.
- We must not store anything into `ptr` and destroy its contents.
- When `ptr` is about to go out of scope, (control exits the block in which `ptr` is defined) we must execute `delete ptr`;



So, here is simple strategy for preventing memory leaks. So, what is the strategy? So, suppose a certain pointer variable `ptr` is the only variable that contains the address of a variable allocated on the heap. So, clearly we must not store anything into `ptr` and destroy its contents. Because that could exactly be a memory leak.

If `ptr` is about to go out of scope, say because control exits a block in which `ptr` is defined we must execute `delete ptr`. So, these are two rules that we have to keep in mind. So, if we keep in mind these two rules then we will be sure that there are no memory leaks.

(Refer Slide Time: 08:11)




Simple strategy for preventing dangling references

Why we get a dangling reference:

- There are two pointers, say `aptr` and `bptr` which point to the same variable on the heap.
- We execute `delete aptr`;
- Later we dereference `bptr`, not realizing the memory it points to has been deallocated.
 - More trivially, if we dereference `aptr` itself.

Simple way to avoid this:

- Ensure that at all times, each variable on the heap will be pointed to only by one pointer!
- More complex strategies are possible. See the book.



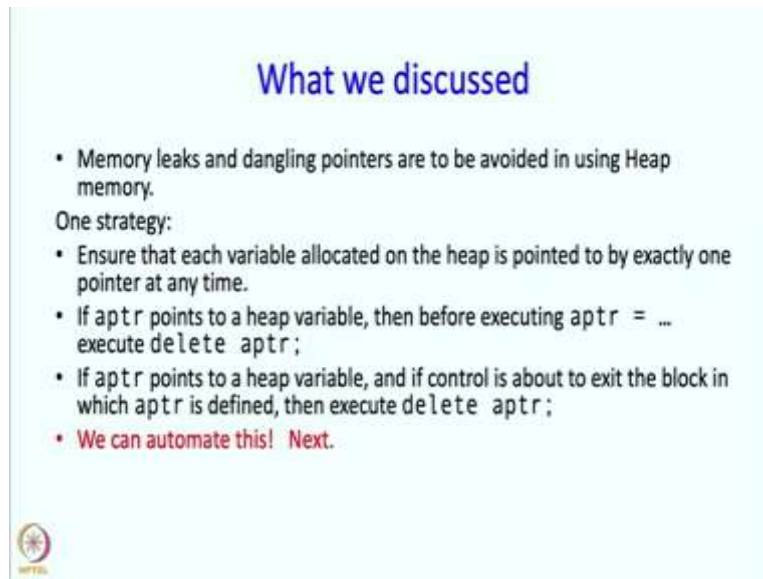
Likewise, there is a simple strategy for preventing dangling references. So, here is sort of reason why we get dangling references. We saw one but this is out of more general thing. So, let us say there are two or more pointers, say just say two for example and let us say they are `aptr` and `bptr` which point to the same variable on the heap and we execute `delete aptr` and we do not realize that `bptr` is also pointing to that same memory and therefore we should not be dereferencing `bptr` either.

So, but suppose we do dereference `bptr` and we get to that memory then that is a dangling reference which we have dereferenced and of course we could have a dereferenced `aptr` itself so both are not good.

So, what should we do? A simple way to avoid this is to ensure that at all times, each variable in the heap will be pointed to only by one ptr. So, there is this case is not going to arise that I deleted the memory using this pointer but I did not realized that some other pointer was also pointing into the same memory but of course, if I deleted the memory using this particular pointer I should also make sure that I do not use this particular pointer and dereference it subsequently.

That is relatively easier to implement of course you can forget but at least that seems easier than saying, 'oh there could be some 5 pointers to the same location on the heap' and do I really know which those 5 pointers are? So that is little bit tricky. So, to avoid that situation we are going to say that at any time each variable on the heap will be pointed only by one pointer and this is not the only possible strategy, more complicated and more efficient and in some ways better strategies are possible, but this a fairly good reasonable strategy.

(Refer Slide Time: 10:32)



The slide is titled "What we discussed" in blue text. It contains a bulleted list of points regarding heap memory management. The first bullet point states that memory leaks and dangling pointers should be avoided. The second bullet point, under the heading "One strategy:", states that each heap variable should be pointed to by exactly one pointer at any time. The third bullet point says that if a pointer 'aptr' points to a heap variable, it should be deleted before being reassigned. The fourth bullet point says that 'aptr' should be deleted when control is about to exit the block where it was defined. The final bullet point, in red, says "We can automate this! Next." There is a small circular logo in the bottom left corner of the slide.

What we discussed

- Memory leaks and dangling pointers are to be avoided in using Heap memory.
- One strategy:
 - Ensure that each variable allocated on the heap is pointed to by exactly one pointer at any time.
 - If `aptr` points to a heap variable, then before executing `aptr = ...` execute `delete aptr;`
 - If `aptr` points to a heap variable, and if control is about to exit the block in which `aptr` is defined, then execute `delete aptr;`
 - We can automate this! Next.

So, what have we discussed? So we have talked about memory leaks and dangling pointers. And we said that to avoid this we should ensure that each variable allocated on the heap is pointed to by exactly one pointer at any time. If `aptr` points to a heap variable then before executing `aptr` equal to something changing its value we execute `delete aptr`, if `aptr` points to a heap variable and if control is about to exit the block in which `aptr` is defined then we execute `delete aptr` as well.

So, this is the strategy and this strategy requires us to remember things but fortunately we can automate it so, that we do not really have to remember all of these things. So, at some point we have to remember and we have to put them into codes such that code will be automatically executed and how that happens, we will talk about in the next segment but before that we will take a quick brake.