Programming Data Structures, Algorithms

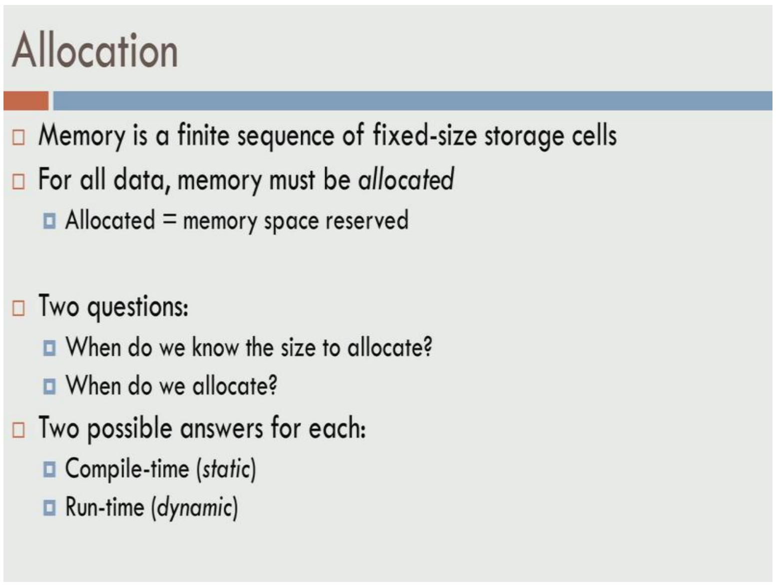
Prof. Shankar Balachandran
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module - 11D
Lecture - 39
Dynamic Memory Allocation
Contents

Hello all, welcome back to this lecture. In this lecture, we are going to look at what is called Dynamic Memory Allocation. So, in all this time so far what we have been looking at is, we know the size of the array is or we know the number of different integers we want or the number of floating points, we want and so on, and that is not always something that you get. So, many times you may want to ask the user, how big the number of elements you want and based on that you want to allocate arrays of an appropriate size and such.

So, in this lecture module what we are going to look at is the notion of dynamic memory allocation. It is so fundamental to various programming languages to be able to allocate memory only on demand. Only if the program needs it, you allocate memory and whenever you do not need it that memory has to be reclaim back and so on. So, to come to this lecture, you should have a reasonable understanding of what pointers are... And if you have not gone through the lecture on pointers, so I strongly suggest that you go and review the lectures on pointers, before you come back here.

(Refer Slide Time: 01:22)



Allocation

- Memory is a finite sequence of fixed-size storage cells
- For all data, memory must be *allocated*
  - Allocated = memory space reserved
- Two questions:
  - When do we know the size to allocate?
  - When do we allocate?
- Two possible answers for each:
  - Compile-time (*static*)
  - Run-time (*dynamic*)

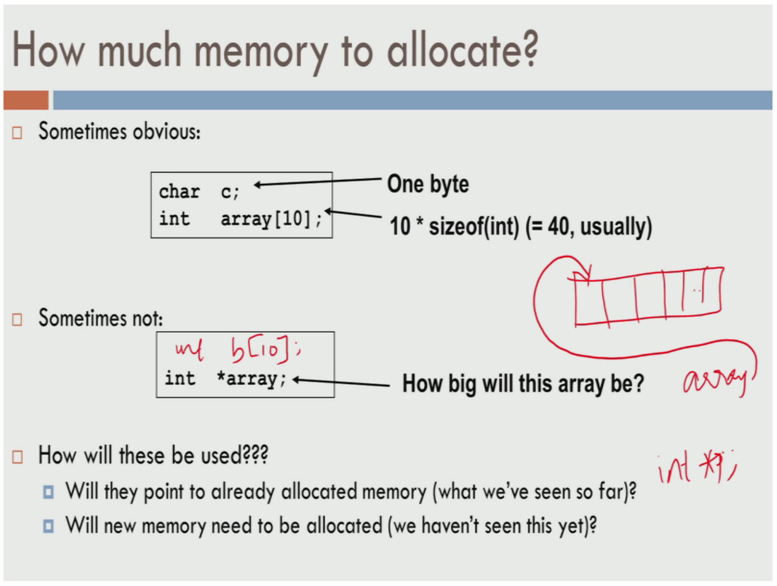
So, let us go back to the very first set of things that I talked about, the notion of allocation of space. So, whenever we declare a variable of any data type, there is allocation of space done to that. So, you take the amount of memory that is there and you go and reserve a certain sequence of bytes and say that this is useful for this variable and no other variable can use this location. This is what we have been doing so far. Whenever I have int x or float y and so on, we assume that there is a particular set of bytes, it is already allocate by the compiler for you to go and use. So, sometimes when you look at these data types, there are two questions that one could ask, and these two are not the same, they are related, but they are not the same. When do we know the size to allocate? So, if I ask you int, if I have int comma int x, do we know the size that we have to allocate, yes if I specify int x I know that I have to reserve space for an integer.

If I do int of a comma a of 10, I want a to be an array of size 10 elements. So, I do know the size that I want. But this is not the same as actually allocating space. So, if I know the space that has to be allocated, I do not have to immediately allocate the space, I can allocate it, when the program starts running. So, I could do it at two possible times, one is called compile time or static allocation, and another is called run time or dynamic allocation.

So, generally if you know the size that has to be allocated, which means if you know statically how much has to be allocated, you would also do the allocation statically.

However, if you do not know the size of the array that you want and so on, you have to wait till the program starts running. When the program starts running, you may have to ask the user how much you want. Based on that you need a mechanism by which you make reservation for only as many as bytes as the programmer wanted. So, that is called dynamic memory allocation. And let us see, how the process goes.

(Refer Slide Time: 03:42)



How much memory to allocate?

- Sometimes obvious:

```
char  c;
int   array[10];
```

One byte

10 * sizeof(int) (= 40, usually)

- Sometimes not:

```
int  b[10];
int  *array;
```

How big will this array be?

- How will these be used???
  - Will they point to already allocated memory (what we've seen so far)?
  - Will new memory need to be allocated (we haven't seen this yet)?

So, let us see this example, character c. The moment a compiler looks at it, it knows that it means only one byte, because characters required only one byte. However, if you have a declaration of size int array of 10, so you need an integer is 4 bytes typically and that into 10 times, you need at least 40 bytes. If the size is known, it is generally also good to go and allocate the space. As soon as the program starts running, you will have 41 bytes allocated, 1 byte for character c and another 40 bytes for array of 0 to array of 9.

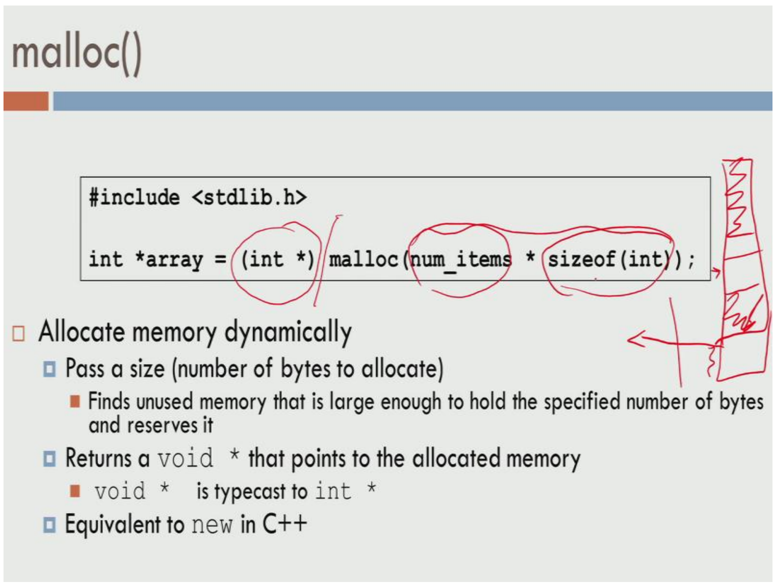
However, sometimes we do not know how much memory to allocate. So, if have int star array, we may not know how big the array is going to be. So, it is a pointer to an integer right now. So, is it pointer to just one integer, is it a pointer to 10 integers, 100 integers, 1 million integers, we do not know that yet. But all we know is, at this point of time we know that it is a pointer to an integer. So, how do we use this?

So, int star array if I have that I could make array point to something that is already allocated. So, I may have something like this. So let us say I had a declaration of this form int b of 10. So, b is already allocated it is space for 10 into 4, 40 bytes at least and I could make array point to b. So, I have b which is of 10 integers and I could make array point to that. So, when you make it point to that it will point to the 0th location of b.

So, in this case allocation is already done, but there are also places, when the declaration is not done. In which case, you have to go and declare or you have to go and claim memory, you have to allocate new memory and claim that space to be yours right now. So, this is something that you probably remember from an earlier lecture. So, when I said something like int star p, p is just a pointer variable, it does not allocate an integer.

It only allocates a pointer which can point to some integer, but the integer itself is not allocated. So, int star array may have to be pointed to something that is already allocated, but if it is already allocated, I should know the size. So, this gets in to a loop, this gets in to a problem of find, knowing the size ahead of the time. So, what we want is a mechanism by which the size may not be known and during the run time of the program and the program already starts running, only then you get to know the size and you want to allocate size is appropriately.

(Refer Slide Time: 06:36)



malloc()

```
#include <stdlib.h>

int *array = (int *) malloc(num_items * sizeof(int));
```

- Allocate memory dynamically
  - Pass a size (number of bytes to allocate)
    - Finds unused memory that is large enough to hold the specified number of bytes and reserves it
  - Returns a void * that points to the allocated memory
    - void * is typecast to int *
  - Equivalent to new in C++

So, to do that there is a function called malloc or m alloc as some people would call it. So, malloc is a function in C and it can take an integer as it is input and it returns a pointer as it is output and this function in this library file called stdlib dot h. So, it is already given as a library to you, you do not have go and write your version of malloc.

So, malloc is a function that is already defined in stdlib for you. What it does is as follows.

You pass a size to it, you pass a number of bytes that you want. Let us say I want 10 integers, each integers is of 4 bytes, I will pass 40 which is 10 times 4 to malloc. What malloc would do is, it would go and scan the memory that is there, find out if there is an unused space. If there is a contiguous chunk of unused space of 40 bytes, it will claim that as a space that you want now and return it to you. So, if I have a huge memory, it is possible that some portions of the memory are already taken and some portions are not.

When malloc is called, let us say I called it with a 40 bytes, I want 40 bytes to be allocated. We will go and looked at the first free space and ask, is there 40 bytes in this space? If the answer is no, it would go and look at the next white space wherever there is no allocation done, is there 40 bytes there. Once, it finds out some free space with 40 bytes which are continuous, it will return a pointer to the caller. So, in this case in this line here, the caller is at this point.

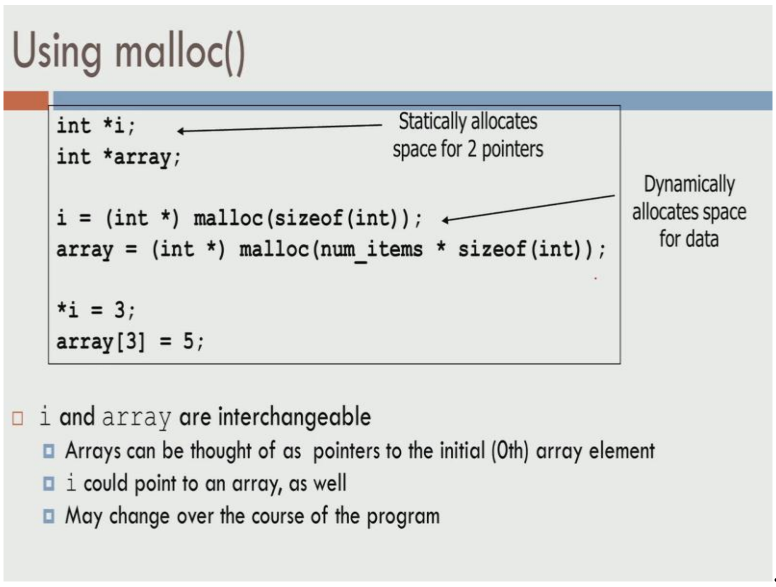
The caller called malloc, malloc would return a pointer and what pointer does it return. It actually returns what is called a void pointer. So, it returns something called a void pointer which means, the data type is not important, I given you space. And then, it is up to you to take that and typecast it to a integer pointer or a character pointer and so on. So, in this case we ask for 10 integers 40 bytes, you will get a pointer to 40 bytes.

So, you make that a data type int star, it now means you have a pointer which is a integer pointer data type. On the left side, you have an integer pointer data type and once you typecast, the right side is also an integer pointer data type, these two are matching. So, you can copy the value of this to the left side. So, now how do we ask for 40 bytes? We did not ask for 40 bytes explicitly, instead we said give me number of items into size of an integer.

So, this is something which is a common thing that you will see in malloc. Instead of actually specifying the number of bytes, you say the number of elements that you want in to size of each element. In this case, num items is the number of items that we want and what we want, we want integers of so many count. So, sizeof is an operator in C, if you pass a variable to it, it will return the number of bytes required. We have already seen this before.

So, this void star in this case is typecasting to int star and this is the way to allocate memory. So, again in summary malloc is a function which is in stdlib, if you pass the number of bytes to it, it will go and scan the memory and find out if there is an unused space of so many bytes. And once it finds out, it will tells that this location is free for you to use, go ahead and use it and you use it as a contiguous space of memory and contiguous space of memory is usually arrays. So, you get a pointer to an array.

(Refer Slide Time: 10:43)



Using malloc()

```
int *i;
int *array;

i = (int *) malloc(sizeof(int));
array = (int *) malloc(num_items * sizeof(int));

*i = 3;
array[3] = 5;
```

Statically allocates space for 2 pointers
Dynamically allocates space for data

- i and array are interchangeable
  - Arrays can be thought of as pointers to the initial (0th) array element
  - i could point to an array, as well
  - May change over the course of the program

.

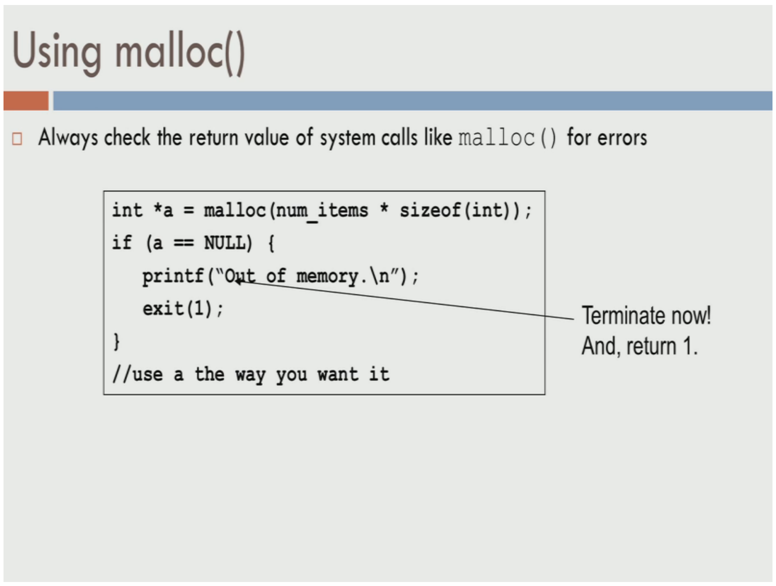
So, let us look at this small example here, so we have two pointers namely, star i and star array. So, i is a pointer of integer type and array is also a pointer to integers, so in this line here, we have i equals int star malloc of sizeof int. So, we are claiming space only for one integer and i is going to point to it. So, i is a pointer to an integer, whereas here we had, as for number of items into sizeof int, you are asking for more than one integer depending on num items.

You get a pointer to so many bytes which can occupy num items integers and you get a pointer to that, that goes into array. So, both i and array are pointers to integers, only that i is pointing only to one integer, whereas array is pointing to a series of locations, in this case num items location. So, array is pointing to num items integers, so you can do star i equals 3, so that will change the contents of the memory location pointed to by i to 3 and you can do things like array of 3 is 5.

So, array of 3 is 5 is valid, because if num items is greater than 4, then array of 3 is valid. So, array of 3 will point to the third location in the array and that is changed to 5. I and array are inter changeable just, because I have 1 integer allocated here and 4 integers or 5 integers allocated here, it does it mean that i and array are incompatible. If we go back to the definition, both are integer pointers. So, i and array are interchangeable.

So, arrays can be thought of as pointers to the 0th element, we have been doing this for a long time now. Whenever I have an array, I have been saying that we can abuse the notion of an array and say that is actually pointed to the 0th location. So, I could also point to an array as well. So, in fact I can do i equals array and i would starts pointing to the array, which array was pointing to earlier and these things can be changed over time. Both the variables, i and array can changed over course of the program.

(Refer Slide Time: 13:11)



Using malloc()

- Always check the return value of system calls like `malloc()` for errors

```
int *a = malloc(num_items * sizeof(int));
if (a == NULL) {
   printf("Out of memory.\n");
   exit(1);
}
//use a the way you want it
```

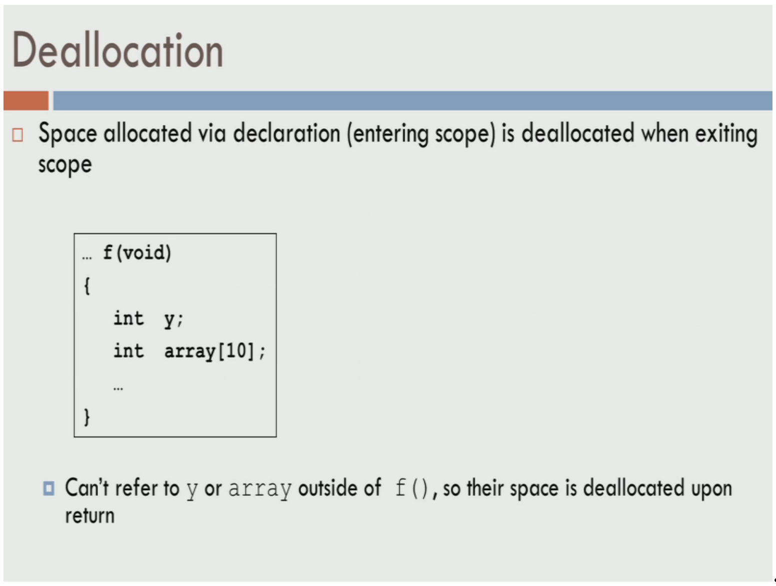
So, one thing that we have to be careful about is I said, if malloc goes and finds out the chunk of space, if that is free it will return the pointer to the chunk of space. What if malloc does not find any free space? You wrote a program which is supposed to process a lot of data and you want more memory allocated to the program, you called malloc. Malloc went and looked at the memory and what of there was no more memory available.

In this case, actually malloc does not return a valid location in the memory, it returns this special case called NULL. We have seen this before, this NULL in capital letters is a special symbol, this cannot be any valid pointer. Because, it is not any valid pointer and malloc if it returns that you know that there is some error that happen. Malloc try to find out your 40 bytes or 100 bytes or 1 million bytes or whatever and if it was not able to find out contiguous chunk of space of so many bytes, it returns with NULL.

So, it is your duty to go and check, if malloc return NULL or not. If it returns NULL, it means some problem happened in memory allocation, you can print it to the user that you actually run out of the memory and you can do exit. So, this exit of one is again a function call, which indicates some kind of an error. So, the moments some problem happens, you may want to exit from the program and say, I did not have enough space, I could not have continued running the program.

But however if a is not NULL, you will not print anything on the screen, you will not call exit. So, a is successfully pointing to some allocation that you did and you can use the a you want it. So, this is something that you are keep in mind. Even though, I am not going to show this thing in all the subsequent code. So, I will not have malloc followed by this check every time, it is something that you have to do every time. You have to go and check, whenever you do a malloc, it is possible that there is something that malloc can not allocate. Therefore, you have to check whether it return NULL or not.

(Refer Slide Time: 15:23)



Deallocation

- Space allocated via declaration (entering scope) is deallocated when exiting scope

```
… f(void)
{
   int  y;
   int  array[10];
   …
}
```

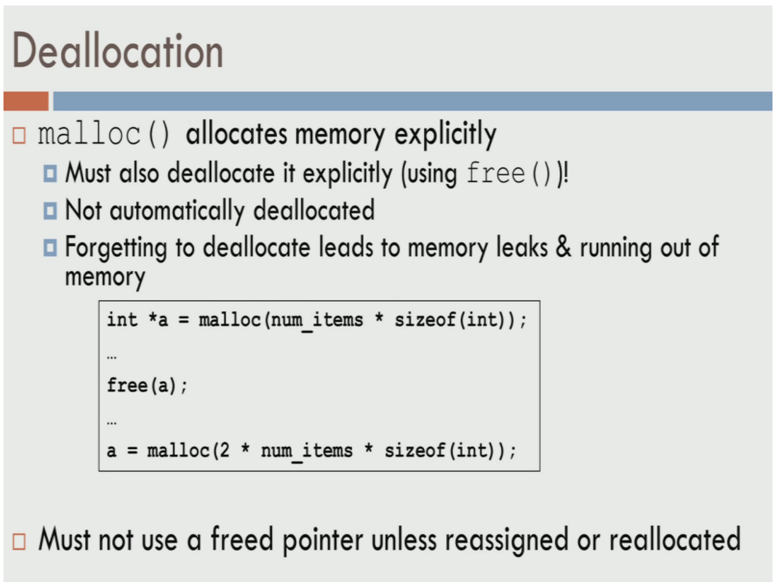
- Can't refer to y or array outside of f(), so their space is deallocated upon return

.

It is also not enough to just allocate memory, you also have to think about what is called deallocation of memory. So, let us go and look at this piece of code here. I have a function f which has int y and array of 10. These are local variables within the function f and so I know this from functions that since is a local variables, they became alive only when the function is called and their automatically killed and returned. When the function returns, they are automatically destroyed. So, we talked about this, when we delete with functions.

So, local variables are good, because when you call the function they became alive and when you return from the function, they became dead. So, when you go to the end of this line f here, y this is an integer is de allocated, array which is an array of 10 integers is also de allocated. You cannot refer to y or you cannot refer to array, the variable array anywhere outside f. This is something that we know already. So, all automatic variables are created automatically and destroyed automatically, that is why they are called automatic variables.

(Refer Slide Time: 16:35)



Deallocation

- malloc() allocates memory explicitly
  - Must also deallocate it explicitly (using free())!
  - Not automatically deallocated
  - Forgetting to deallocate leads to memory leaks & running out of memory

```
int *a = malloc(num_items * sizeof(int));
…
free(a);
…
a = malloc(2 * num_items * sizeof(int));
```

- Must not use a freed pointer unless reassigned or reallocated

Whereas, if you do call something with malloc, you are getting a pointer to the memory that was pointed to and it is your duty now, you claim the memory you wanted some memory and you have to free it out. So, it has to be done using this function called free and forgetting to de allocate. So, let us say I claim some memory and I do not have used for it anymore, I have to de allocate it and for that I will use free. So, for example here int star a is malloc, so many bytes as num item integers demands and I used that.

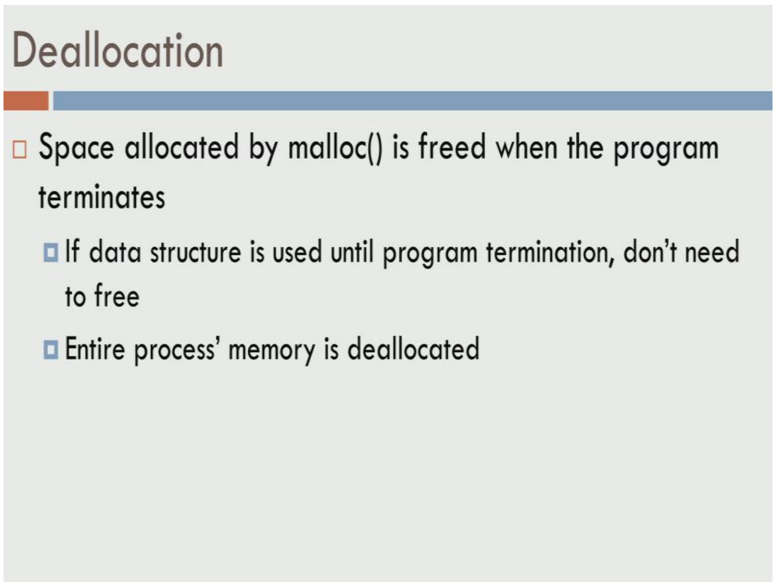
If I do not have any use for it, I should go and free up a. So, if you do not do that what might happen is that this chunk that you allocated may remain as both. Somebody claimed it and somebody made a reservation and it is still been used, that is how the complier would treated. So, this is like you go to a hotel, let us say you go to a restaurant you make a phone call and you reserve a table. You reserved the table, but you never turned up.

Then, the restaurant would have to know at some point, whether somebody is going to turn up and use the table or not. And if you are not going to call, if you are not go to the restaurant, it is good for you to call the restaurant and tell them, I even though I called, I made a reservation I do not want it any more. So, that is equivalent to free or you actually went to the restaurant, you had your meal you return back and that point it is not reserved anymore. So, you have to say that this table is not reserved any more, it has to be freed again.

So, this is the equivalent or analogy that I can give you for malloc and free. Whenever you claim some space, it is your duty to go back and tell the system to reclaim it. You do not need this anymore. So, once you free it, you cannot use a anymore below this line. So, once I have freed a I do not have space reserved for a anymore, a is not pointing to anything. Therefore, I cannot say a of i equals 5 and so on, i cannot do that.

I should either go and do new allocation or I should make a point to something that is already allocated. So, I could do something like a equals b, if b was declared as int b of 10 or I could do new allocation as it was given here. So, you must not use a free pointer, unless it is re assigned or re allocated. So, keep this in mind.

(Refer Slide Time: 19:19)

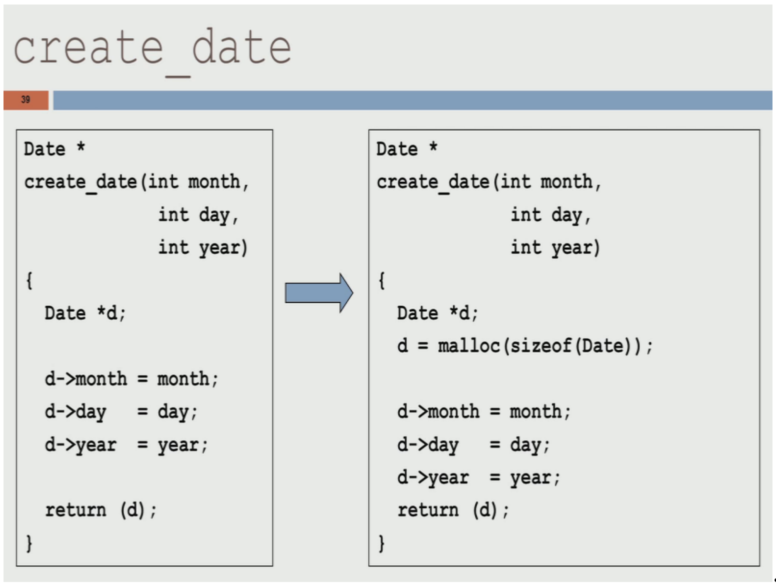


Deallocation

- Space allocated by malloc() is freed when the program terminates
  - If data structure is used until program termination, don't need to free
  - Entire process' memory is deallocated

So, space allocated by malloc is freed when the program terminates. So, if you do not free of memory ever, so you allocate some memory, you use it. But let us say you do not do free and you keep doing that. So, slowly and steadily your memory will get used up, at some point of time your memory might become full. So, in any of these cases if you do not do explicit memory de allocation, when the program exits, your program is done with it is work, you do not need to claim this space anymore. Automatically, everything gets de allocated by your operating system. So, you can wait for your program to terminate and free up all memory that your program claim. So, some people do that, but that is not a good practice. So, the good practice is assume when your program does not need some memory as well deallocated, because your own program may need more memory for doing some more work later.

(Refer Slide Time: 20:20)



create_date

39

```
Date *
create_date(int month,
             int day,
             int year)
{
  Date *d;

  d->month = month;
  d->day   = day;
  d->year  = year;

  return (d);
}
```

```
Date *
create_date(int month,
             int day,
             int year)
{
  Date *d;
  d = malloc(sizeof(Date));

  d->month = month;
  d->day   = day;
  d->year  = year;
  return (d);
}
```

.

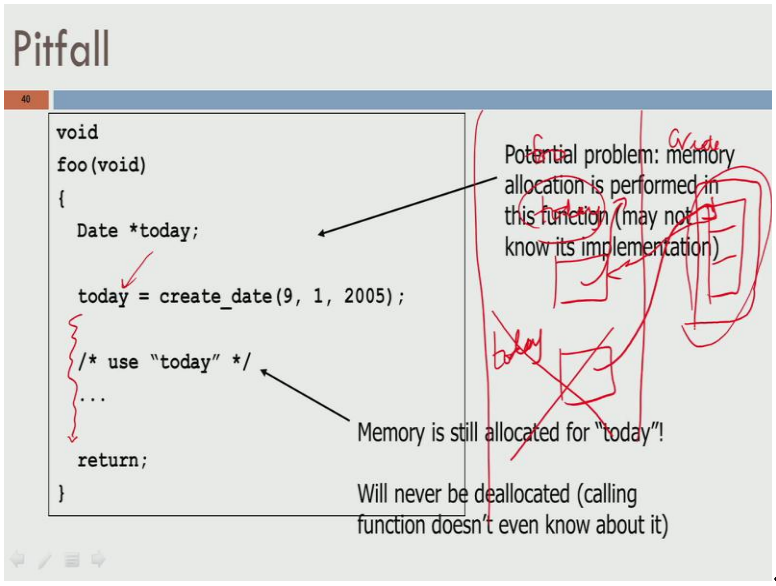
So, let us look at a small function called create date. It is going to take three integers month, day and year and it is supposed to return a date. So, let us say I did this, Date star d. So, d is a pointer of the type Date and remember, it does not allocate three things for month, day and year. It only declares a pointer to type Date. So, you have a space for a pointer, but the actual members are not allocated yet and let us say you did this.

D is month, it is supposed to be the month that you passed on, d is day it is supposed to be the day that you passed on and d is year, it is supposed to be the year that you passed on and return the pointer d, let us say we did this. So, this should be wrong, because d is not allocated any memory. So, the correct way to do that is this follows. D is a pointer, first allocates space and how much space do you want to allocate?

I want to allocate space for one Date type and what does the Date type have? It has one month field, one day field and one year field, we need three things. So, when you see sizeof Date, sizeof Date is three integers, one for month, one for day and one for year. And when you do malloc of sizeof Date that returns a pointer that is assigned to d. So, now d is actually pointing to an allocated space. Once it is pointing to an allocated space, it is to do d arrow month, d arrow day and d arrow year.

And when you return d, you are actually returning a pointer to the reserved space. So, the reserved spaces still exist and you are returning the pointer to the reserved space. So, the function on the right side is correct, whereas the function on the left side is not. Because, you did not allocate space on the left side, you explicitly allocated space on the right side. So, one thing we have to do is even though the space is explicitly allocated, remember create date is not responsible for de allocating it. You are not explicitly de allocating it yet. You are only explicitly allocated it. It is somebody's job to come and de allocated later, we will see who is this somebody must be later.

(Refer Slide Time: 22:38)



Pitfall

```
void
foo(void)
{
  Date *today;

  today = create_date(9, 1, 2005);

  /* use "today" */
  ...

  return;
}
```

Potential problem: memory allocation is performed in this function (may not know its implementation)

Memory is still allocated for "today"!

Will never be deallocated (calling function doesn't even know about it)

.

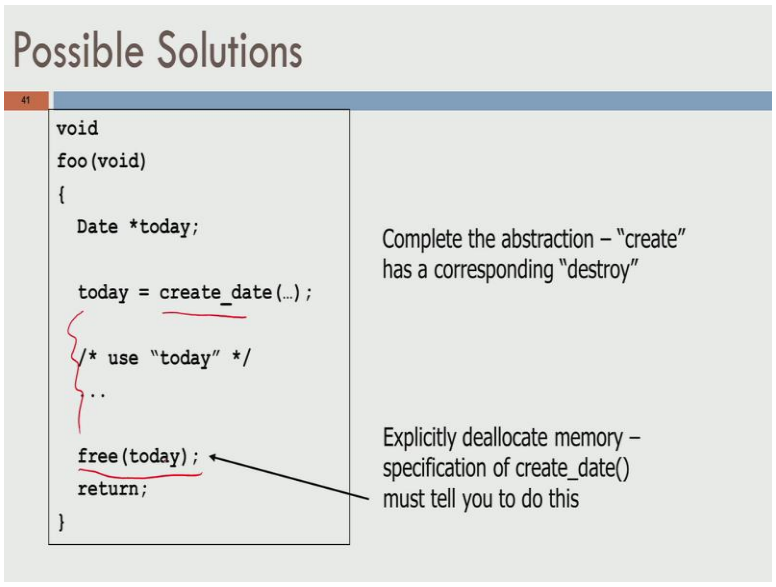
Let us see a small thing here. So, Date star today, today is create date 9, 1, 2005 and do return. So, I want to do let us say 9th of January 2005. I created a variable called today. So, today is a pointer to the data type called Date, create date in turn returns a pointer to data type called date. I used today and then I returned it. So, at this point create date allocated some memory and you got that in today. So, let us look at the sequence of things.

So, inside this function called foo, so today is a local variable and what is today, it is actually a pointer. It is supposed to point to a Date data type. And when you call create date, so inside create date your allocated space one for date, one for month and one for year, you allocated space for the structure, you return the pointer to this. So, at this point your today field is pointing to the location that you gave for the structure. And let us say, we use today and we return, we came to the end of it.

And this point we return from foo, what would happen now? Since, today is a local variable within foo, today gets destroyed. It is an automatic variable, the automatic variable gets destroyed. So, whoever called foo cannot access today, because today is a local variable within foo. And because today is lost, you also lost a pointer to some reserved space that you had. You reserved space for a structure, but today which was holding the pointer to the structure was destroyed.

Therefore, you lost the trail to go back to the memory that you allocated, so this is the problem. So, whenever you have something that is allocated from a function and if you put that in the local variable and if you did not explicitly de allocated, that thing is still going to look around. This structure that you had is still going to look around and you will have no way to free it, this will get freed only when the program gets done. So, you have to be careful about it.

(Refer Slide Time: 25:17)



Possible Solutions

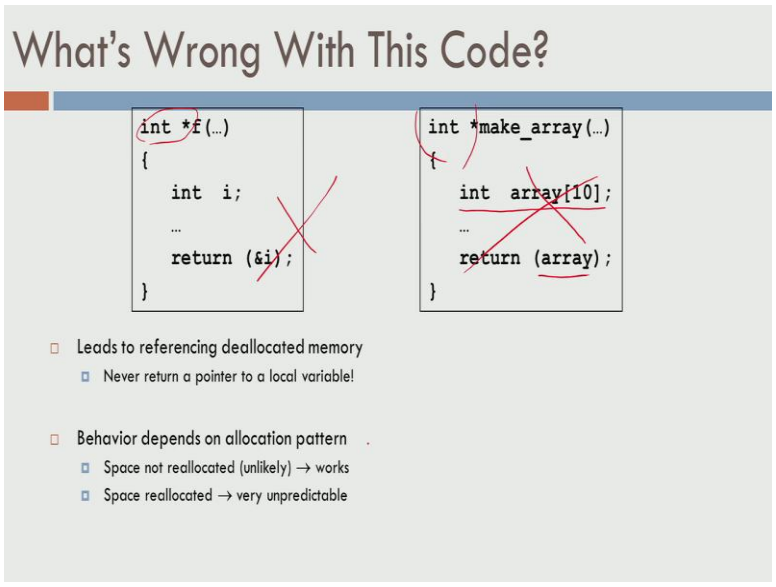
```
void
foo(void)
{
  Date *today;

  today = create_date(…);

  /* use "today" */
  ..

  free(today);
  return;
}
```

Complete the abstraction – "create" has a corresponding "destroy"

Explicitly deallocate memory – specification of create_date() must tell you to do this

So, there are various possible solutions and the cleanest solution is, let us say, I did create date. I did whatever I want to do with Date, I know that I do not need date anymore and I am returning from the function as well free today. So, that when you say free today, the function will go and looked at the space allocated are reserved for it and remove the reservation. So, that space can be used by the program, some other part of the program. So, this free takes care of that and when you return it, you already taken care of whatever allocation you have done, you have removed your reservation.

(Refer Slide Time: 25:53)



What's Wrong With This Code?

```
int *f(…)
{
    int  i;
    …
    return  (&i);
}
```

```
int *make_array(…)
{
    int  array[10];
    …
    return  (array);
}
```

- Leads to referencing deallocated memory
  - Never return a pointer to a local variable!
- Behavior depends on allocation pattern
  - Space not reallocated (unlikely) → works
  - Space reallocated → very unpredictable

So, now I am going to show you a sequence of various things and I want to argue about, why something is correct or why something is incorrect. So, let us look at the function on the left side. So, int star f, so f is a function and i is a local variable within the function and you are returning address to the local variable. So, that is what this int star does. So, int star means you are returning a pointer to an integer. Is this something that you can do?

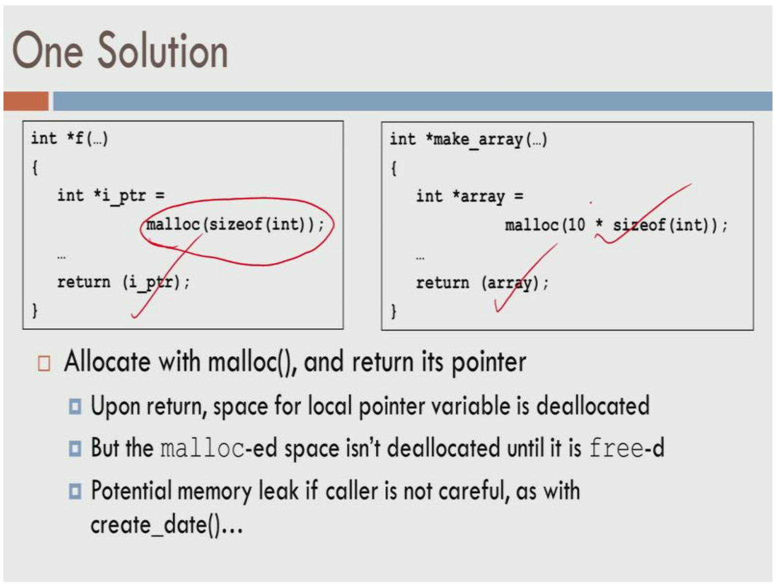
This is not possible, because when the function gets return i is a local variable, i gets destroyed, it is not correct to track the address of i anymore. So, this is not correct. It is de allocated i, you cannot track it anymore. Let us look at this function on the right side, it is also supposed to return an integer star, it is returning a pointer to an integer. So, here you have a local variable called array, which is an array of 10 integers and you return array which means, you are returning the location of the 0th element of array.

But again this return array is supposed to return a pointer to the 0th location. But since array is local to make array, array gets destroyed when you return from make array. So, you are having a pointer that is actually pointing to a location that you cannot claim to be yours anymore. So, this is a problem, so if you do this, sometimes it may look like this whole thing is actually working. So, this might work, because when you return back from this make array, may be the space is still containing all the values, nobody reclaimed it.

So, when you go back to that location, you still see the values. But if you are unfortunate, this memory location that you did not want anymore got used up somewhere else and now you are actually accessing something that is not supposed to be used by you. So, the space if it is reallocated this becomes unpredictable, but if the space is not reallocated to anyone else, it might work. In general, it should not rely on the space to be unallocated.

You should not believe that I am hoping that it is unallocated, let me go and use it. That is not correct. You should treat, you should be conservative and you should say that I will assume that it is going to be reallocated. Therefore, I will free it up here and I will not pass pointers back to local variables.

(Refer Slide Time: 28:24)



One Solution

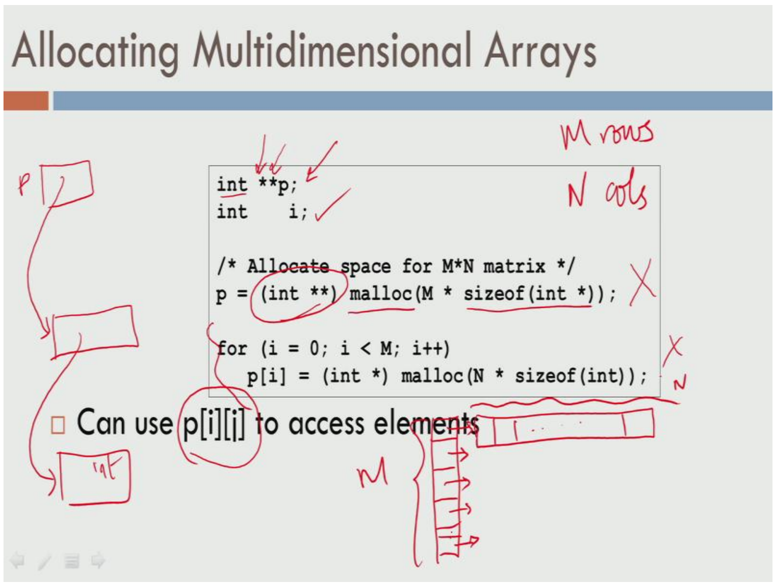
```
int *f(…)
{
   int *i_ptr =
            malloc(sizeof(int));
   …
   return (i_ptr);
}
```

```
int *make_array(…)
{
   int *array =
            malloc(10 * sizeof(int));
   …
   return (array);
}
```

- Allocate with malloc(), and return its pointer
  - Upon return, space for local pointer variable is deallocated
  - But the malloc-ed space isn't deallocated until it is free-d
  - Potential memory leak if caller is not careful, as with create_date()…

So, if you want to really pass pointers to local variables or anything that you allocate inside, you first of all do dynamic allocation and you pass on a pointer. This is correct. Similarly, here you did dynamic allocation, so these 10 integers are not going to be de allocated when the function returns and because of that it is to return a pointer to that. So, you allocate with malloc and return the pointer, so this is malloc.

(Refer Slide Time: 28:53)


Allocating Multidimensional Arrays

```
int **p;
int   i;

/* Allocate space for M*N matrix */
p = (int **) malloc(M * sizeof(int *));

for (i = 0; i < M; i++)
   p[i] = (int *) malloc(N * sizeof(int));
```

- Can use p[i][j] to access elements

So, now let us go and look at, how to allocate multidimensional arrays. It is not just that we looked for single dimensional arrays or array of integers, array of characters and so on. Many times you want to build matrixes or 3D matrixes and so on, how do we do it. So, we starts with int star star p. So, the meaning of that is p is a pointer to a pointer to an integer. So, let us see what that means. So, p is not pointing to an integer, p in term is pointing to another pointer which is in turn pointing to some integer. So, that is the meaning of int star star p.

So, p is supposed to be pointing to another pointer which in turn pointing to an integer. So, int i is declaring space for an integer i. Now, let us see how to allocate a multidimensional array. So, the first thing you do is let us say I want M rows and N columns. So, the first thing you do is allocate M pointers first. So, we have M pointers that are being allocated and this is not going to get pointer to anything.

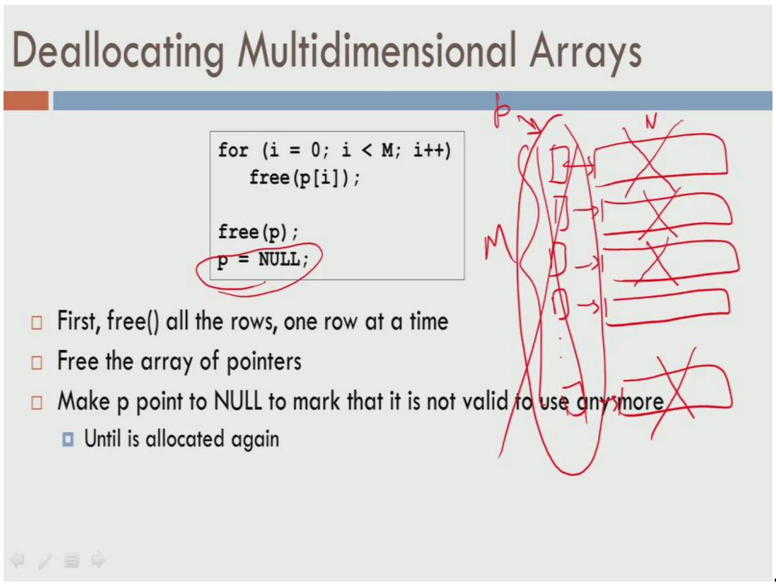
So, you half M pointers, we do that first and then each pointer variable, you now make an allocation of N integers. So, you allocate M pointers to integers and then, you allocate N integers at a time and this is supposed to give you N integers. So, each row you get from malloc and you have M rows that gives you M cross N matrix. So, let us break this down, the first thing we have done is a malloc of sizeof int star. So, sizeof int star is sizeof integer pointer, not sizeof integer it is sizeof integer pointer and we want M integer pointers.

Let us say you got them, you got M integer pointers and they are not going to point to anything, because malloc does not initialize. So, what you get in return? Malloc will return a pointer to 10, let us say 10 rows. We will get pointers to an array of pointers and these are pointing to 10 integer arrays. So, an int star star tells you that the left side is expecting a pointer to a pointer. It is not a pointer to an integer. It is pointer to a pointer to an integer.

And once you have that you have all these pointers, which are ready to now point to the ten different rows, but you have not allocated space for the rows yet and that is done inside a loop. So, this loop goes from i equals 0 to M minus 1 and you allocating one row at a time and each row is of size n integers. So, that is what you have here. So, this is the very basic way in which M rows by N columns is allocated. At the end, you actually have M and N elements.

In fact, once you have this you can use p of i j to access the elements. So, just like you would access array of i comma j, two dimensional array of i comma j, p is a pointer to a pointer to an integer. You can use p of i comma j to get to the basic elements. So, it would be incorrect to access p of i j, if you have not done this or if you have not done this. You should have done both these mallocs, so you should have malloc for 10 pointers to integers and you should have allocated for each pointer to point to a row of N numbers. So, that will give us 10 rows by N columns.

(Refer Slide time: 33:06)



Deallocating Multidimensional Arrays

```
for (i = 0; i < M; i++)
   free(p[i]);

free(p);
p = NULL;
```

- First, free() all the rows, one row at a time
- Free the array of pointers
- Make p point to NULL to mark that it is not valid to use any more
  - Until is allocated again

.

So, let us look at how to de allocate multidimensional arrays. So, right now you have let us say 10 pointers pointing to one row each and each row, it is of size N elements. So, you have M of these and you have N of those. What you do is, you first go and do free of p of i and you iterate i from 0 to M minus 1. So, when you say free p of i, you start with p of 0, free p of 0 will de allocate that. Free p of 1 will de allocate this row, p of 2 being free will de allocate this row and so on, free p of M minus 1 will de allocate this row.

So, you have done with de allocating all the integers, but you still have all these pointers that you allocated and the way you allocated that was p was pointing to all these 10 things as though, they were an array of pointers. And when you do free of p, it also de allocates all the pointers. At this point, the pointer p is not pointing to anything valid, you make p equals NULL. So, essentially what you have is you have a sequence of steps.

When you want to allocate two dimensional arrays, you first go and allocate a pointer to a pointer to the basic data type and you iterate over that one at a time and allocate rows. When you de allocate, you de allocate all the rows one at a time and then you de allocate the array of pointers itself. So, this is the way you allocate and de alocate pointers. So, first free all the rows, free the array of pointers, make p point to the NULL and you have done.

So, this is p will keep pointing to NULL, until there is new allocation. So, with this we are end up at this lecture about pointers and how to use dynamic memory allocation for not just 1D arrays, it can also be 2D arrays. The basic element that you stored in an array can be an integer, can be a float, it can be a character, it can also be a structures. You can have a 2D matrix of structures, where each element could be an integer containing x, an integer containing y may be you have a 2D array of pointers that is possible.

So, to summarise what we have is we have a mechanism, a general mechanism by which as long as I know the size of the basic data type that I want, I will allocate bytes as many as I need. I will get a pointer returned by malloc, I should point it to the appropriate data type and whenever these memories not needed anymore, you should be really careful and de allocated the memory. So, that your memory is not hack, your program itself further done may need more memory, if you did not free of something that you no need, you may get struck. So, this brings us to the end of the lecture.

Thank you.