Programming in Modern C++

Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture 10

Dynamic Memory Management

(Refer Slide Time: 00:35)

Programming in Modern C++

Module M10: Dynamic Memory Management

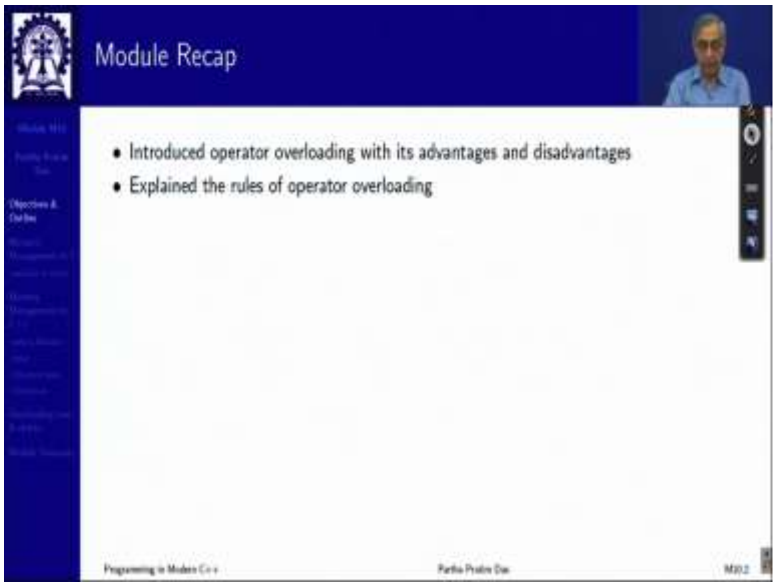
Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Programming in Modern C++ Partha Pratim Das M10.1



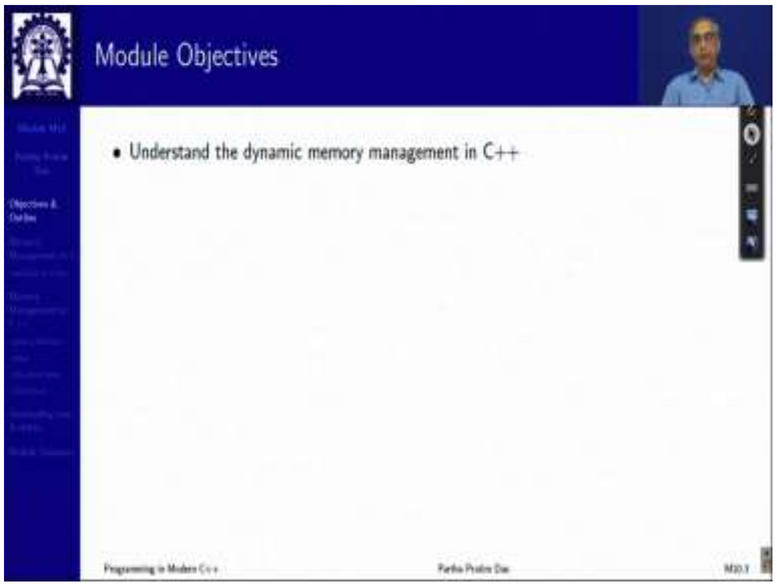
Module Recap

- Introduced operator overloading with its advantages and disadvantages
- Explained the rules of operator overloading

Programming in Modern C++ Partha Pratim Das M10.2

Welcome to programming in modern C++. We are in week 2 and we are going to discuss module 10. In the last module we talked about ADHOC polymorphism of operator overloading, which is a very key concept extension over C++, it allows operators to be treated as operator functions and allows them to be overloaded in the context of it enum or a structure, it cannot be overloading cannot be done for the built in types but for others it is possible and we discuss the rules for that.

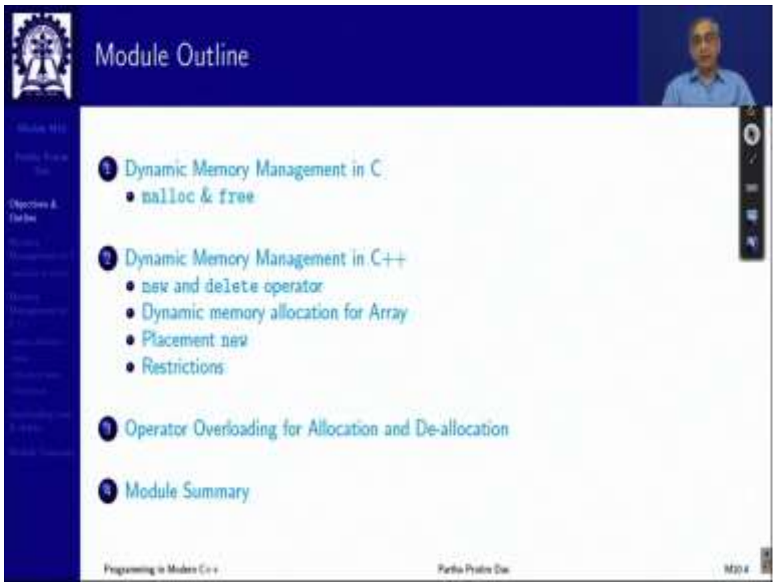
(Refer Slide Time: 01:09)



Module Objectives

- Understand the dynamic memory management in C++

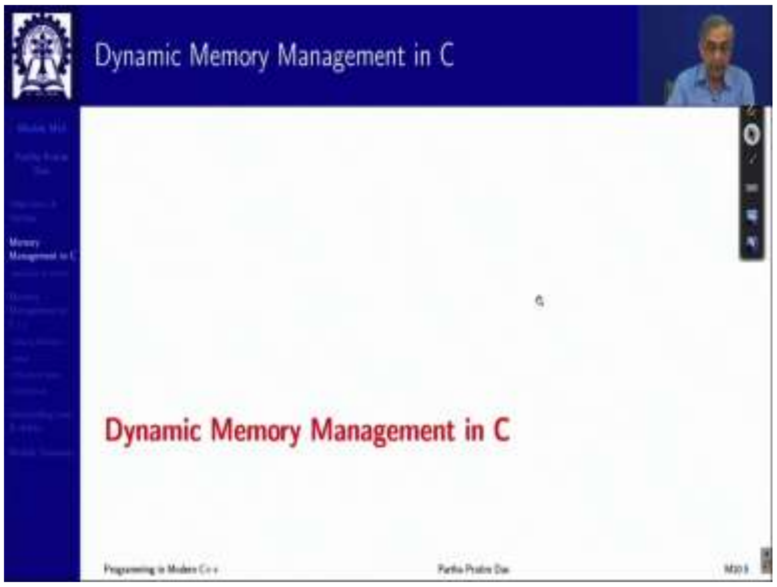
Programming in Modern C++ Partha Pratim Das M10.3



Module Outline

1. Dynamic Memory Management in C
   - `malloc` & `free`
2. Dynamic Memory Management in C++
   - `new` and `delete` operator
   - Dynamic memory allocation for Array
   - Placement `new`
   - Restrictions
3. Operator Overloading for Allocation and De-allocation
4. Module Summary

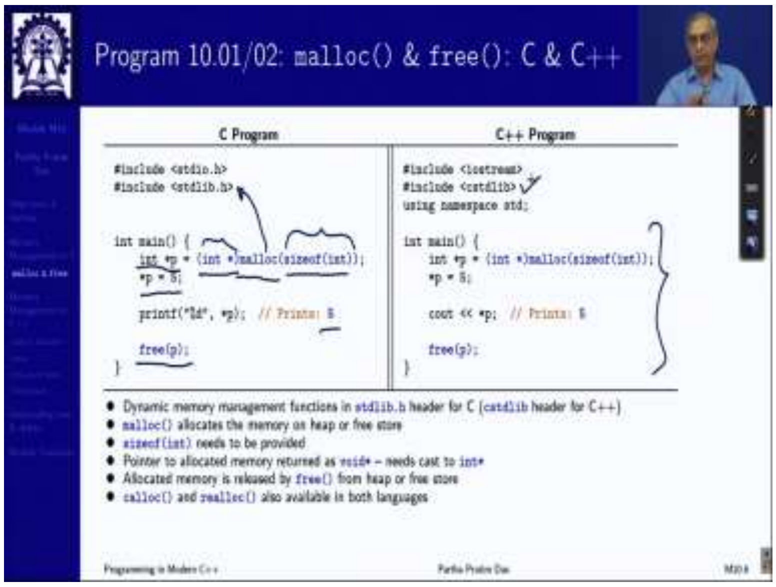
Programming in Modern C++ Partha Pratim Das M10.4



Dynamic Memory Management in C

Dynamic Memory Management in C

Programming in Modern C++ Partha Pratim Das M10.5



Program 10.01/02: malloc() & free(): C & C++

C Program

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;

    printf("%d", *p); // Prints: 5

    free(p);
}
```

C++ Program

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;

    cout << *p; // Prints: 5

    free(p);
}
```

- Dynamic memory management functions in stdlib.h header for C (cstdlib header for C++)
- malloc() allocates the memory on heap or free store
- sizeof(int) needs to be provided
- Pointer to allocated memory returned as void* – needs cast to int*
- Allocated memory is released by free() from heap or free store
- calloc() and realloc() also available in both languages

Programming in Modern C++ Partha Pratim Das M10.6

We will again take up another very core area today in this module which is dynamic memory management in C++. And I know you all are familiar with the dynamic memory management in C using malloc and free, so we will start with that and then talk about how to do these things in C++ and how to do them better. This is the overall outline you find always on left.

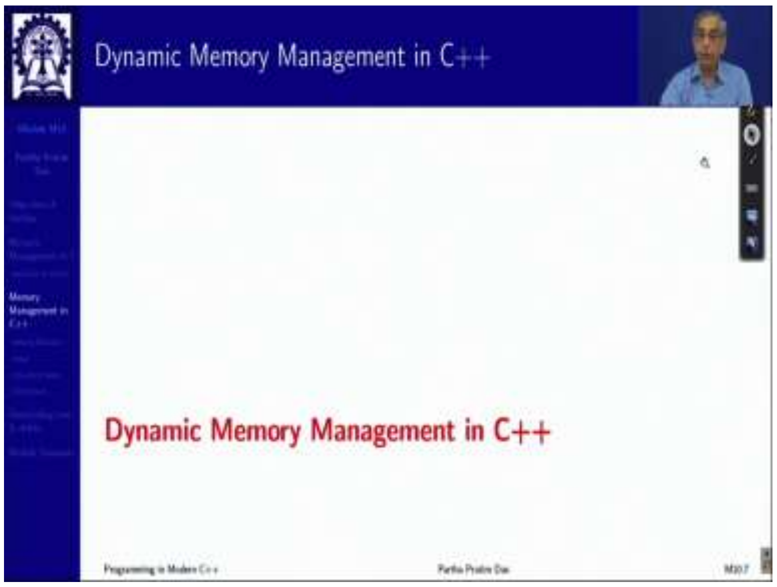
So dynamic memory management in C, I am sure all of you know that you can dynamically allocate memory at the runtime, a memory of a basic contiguous size through the use of malloc function, memory alloc function where you pass the number of bytes that you want in the allocation. The allocation is done in the free store or heap as we refer to it and the allocation is contiguous.

malloc is a function available in this standard library, so you need to include stdlib.h. malloc, once malloc allocates it returns you a pointer which is of void type. So, if you are allocating an integer which is what is obvious here because I am passing the sizeof an integer, if it is four bytes then I am passing four.

Then malloc returns a void* pointer which needs to be cast to the integer pointer. malloc will always return void* so I cast it to integer pointer and p is now a pointer to this dynamically allocated integer. By dereferencing, I put a, assign a value 5 to that I print it I can see. Once I am done I can free this memory up back to the system by calling free on this pointer p.

Exactly the same code can be written in C++ only you need to change the naming of the header. There are other functions in C also like you have calloc, you have realloc and all that, all those are also similarly available in C++, we will not go into details because you already know all this, I just gave this example as a starting point for the discussion.

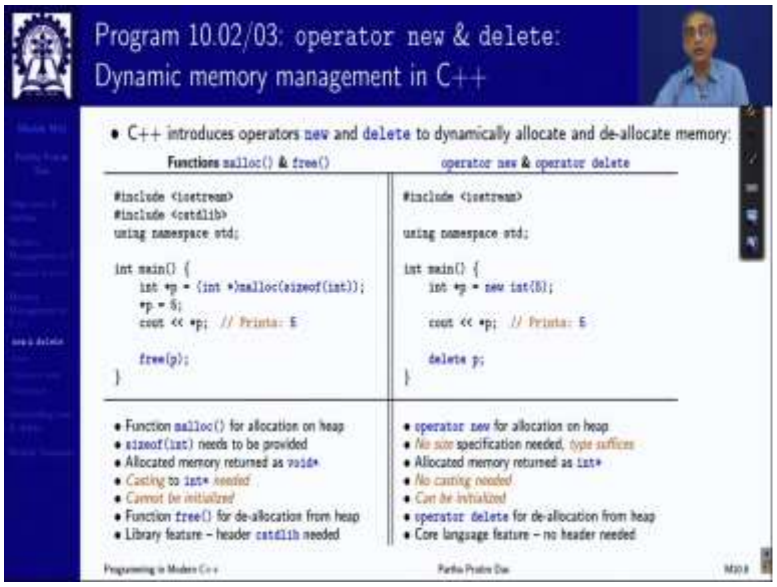
(Refer Slide Time: 03:54)



Dynamic Memory Management in C++

Dynamic Memory Management in C++

Programming in Modern C++ Partha Pratim Das M10.7



Program 10.02/03: operator new & delete: Dynamic memory management in C++

- C++ introduces operators new and delete to dynamically allocate and de-allocate memory:

Functions malloc() & free()	operator new & operator delete

`#include <iostream>`
`#include <cstdlib>`
`using namespace std;`
 	
`int main() {`
`    int *p = (int *)malloc(sizeof(int));`
`    *p = 5;`
`    cout << *p; // Prints: 5`
 	
`    free(p);`
`}`
	`#include <iostream>`
	
	`using namespace std;`
	
	`int main() {`
	`    int *p = new int(5);`
	
	`    cout << *p; // Prints: 5`
	
	`    delete p;`
	`}`

• Function malloc() for allocation on heap • sizeof(int) needs to be provided • Allocated memory returned as void* • Casting to int* needed • Cannot be initialized • Function free() for de-allocation from heap • Library feature – header cstdlib needed	• operator new for allocation on heap • No size specification needed, type suffices • Allocated memory returned as int* • No casting needed • Can be initialized • operator delete for de-allocation from heap • Core language feature – no header needed

Programming in Modern C++ Partha Pratim Das M10.8

So let me jump into the memory management, dynamic memory management in C++. So this is the C scenario you use malloc, pass the size, cast it to the appropriate type, the pointer type, and free to free up the memory. In C++ you have a new operator new, it is a new addition. So new is an operator, like sizeof is an operator, new is an operator.

Very interestingly the operand of new is not a value it is a type, so new is an operator, is a prefix operator comes first. Its first parameter which is mandatory, you cannot have new without this first parameter, is not a variable or a value it is a type, very, very interesting. Then it has optional other parameters here this parameter means the initialization value.

So, what it does, it does the same thing as very similar thing as malloc, let me not say same thing as malloc, similar thing as malloc. A space is allocated which is adequate to hold an integer, five is placed in that initialized here, not assigned later on, it is initialized with five and then you get a int* pointer returned, which you are initializing p with.

So some key differences, so the key similarities both of them malloc and operator new allocate on heap, but the key difference is malloc is type oblivious, does not understand the type, so malloc has to be categorically told give me so many bytes that is all, you do sizeof and do that, but operator new is type conscious that its first parameter.

So since it knows the type it does not need to be given the size because it can compute internally if it is int sizeof int it knows, so it knows how many bytes it has to do. Similarly, when malloc returns, malloc is returning you an array of bytes, so there it is again type oblivious so it returns a void pointer but here new knows the type, so you returns you a pointer of that type.

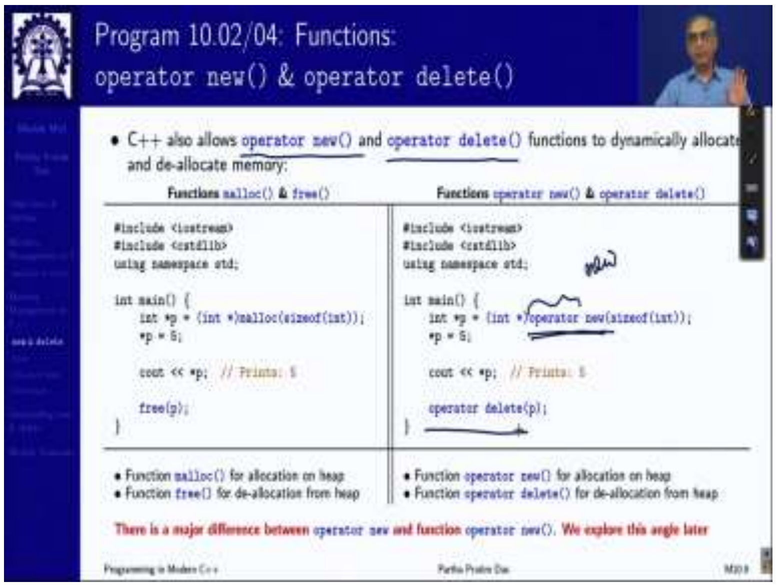
So, this int, the type is used in two places one to compute the size other to return a pointer of that same type, no casting is required. Further new can take additional parameter like as we have given here as initialization, so the value is not only created in the free store, but is also the variable is also initialized nothing like that can be done for malloc.

Now obviously corresponding to new there has to be a similar operator to release the memory and that is called delete. Again delete is an operator, again delete is a prefix operator but unlike new it does not need to take any type because it takes the pointer that it has to release and the type of the pointer, so what to do with that how much size was there and how much to be released and so on.

So that is a parallel of free where free also works with the pointer, but it treats it simply as a byte array. There is a deeper question as to delete would be able to know how many bytes to free up release because delete knows the type p, free is again type oblivious it is in C, so the parameter of free is void*, so you can pass p you can actually pass it like this also that will also work. Then the question is how does free get to know what is the size?

So there has to be some mechanism we will talk about it at some point of time. Finally the malloc and free are C standard library support whereas new and delete are operators so their language support. So in C++ memory management becomes a core of it which was dynamic memory management becomes a core part of the language which was not there in C.

(Refer Slide Time: 09:33)



Program 10.02/04: Functions: operator new() & operator delete()

- C++ also allows operator new() and operator delete() functions to dynamically allocate and de-allocate memory:

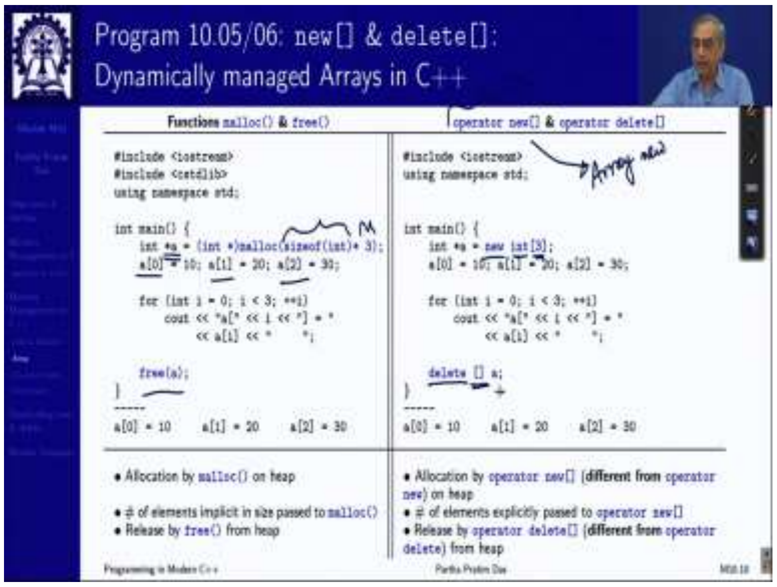
Functions malloc() & free()	Functions operator new() & operator delete()

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;

    cout << *p;  // Prints: 5

    free(p);
}
```
	```
	#include <iostream>
	#include <cstdlib>
	using namespace std;

	int main() {
	    int *p = (int *)operator new(sizeof(int));
	    *p = 5;

	    cout << *p;  // Prints: 5

	    operator delete(p);
	}
	```

• Function malloc() for allocation on heap • Function free() for de-allocation from heap	• Function operator new() for allocation on heap • Function operator delete() for de-allocation from heap

There is a major difference between operator new and function operator new(). We explore this angle later

Programming in Modern C++ Partha Pratim Das M10.9



Program 10.05/06: new[] & delete[]: Dynamically managed Arrays in C++

Functions malloc() & free()	operator new[] & operator delete[]

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main() {
    int *a = (int *)malloc(sizeof(int)* 3);
    a[0] = 10; a[1] = 20; a[2] = 30;

    for (int i = 0; i < 3; ++i)
        cout << "a[" << i << "] = "
             << a[i] << "    ";

    free(a);
}
-----
a[0] = 10     a[1] = 20     a[2] = 30
```
	```
	#include <iostream>
	using namespace std;

	int main() {
	    int *a = new int[3];
	    a[0] = 10; a[1] = 20; a[2] = 30;

	    for (int i = 0; i < 3; ++i)
	        cout << "a[" << i << "] = "
	             << a[i] << "    ";

	    delete [] a;
	}
	-----
	a[0] = 10     a[1] = 20     a[2] = 30
	```

• Allocation by malloc() on heap • # of elements implicit in size passed to malloc() • Release by free() from heap	• Allocation by operator new[] (different from operator new) on heap • # of elements explicitly passed to operator new[] • Release by operator delete[] (different from operator delete) from heap

Programming in Modern C++ Partha Pratim Das M10.10

So having said that let me again introduce you to something interesting as well as little confusing, do not get confused I am telling you please do not. It also has C++ also has two functions called operator new and operator delete, very confusing. When you say new it is an operator when you say operator new it is a function.

So when you use this and this is mind you this is not the operator function of new, operator function of new also will be there but this is a separate function operator new it is his name and since its name is operator new you need to write operator here. If you simply write nu it means the operator nu not this function, so what the operator new function does?

It behaves like malloc. So it is a kind of you can say it is a mimic for malloc it takes a size allocates the number of bytes in the free store and returns a void* pointer. Similarly there is operator delete which takes a void* pointer returns a void like free, so these are corresponding ones. So that actually in C++ you should not require to use any of the C dynamic memory management functions.

Now the question is what if you have to allocate memory for arrays? What do you do in C? In C there is only one way you can have a byte array, so you put element size, you put you the number of elements here multiply that is a total byte size and get that allocated get a void* pointer cast it to the array type and because a pointer is an array, so when you get it as pointed to int you can use it as a[0], a[1], a[2], like that as an array free is similar.

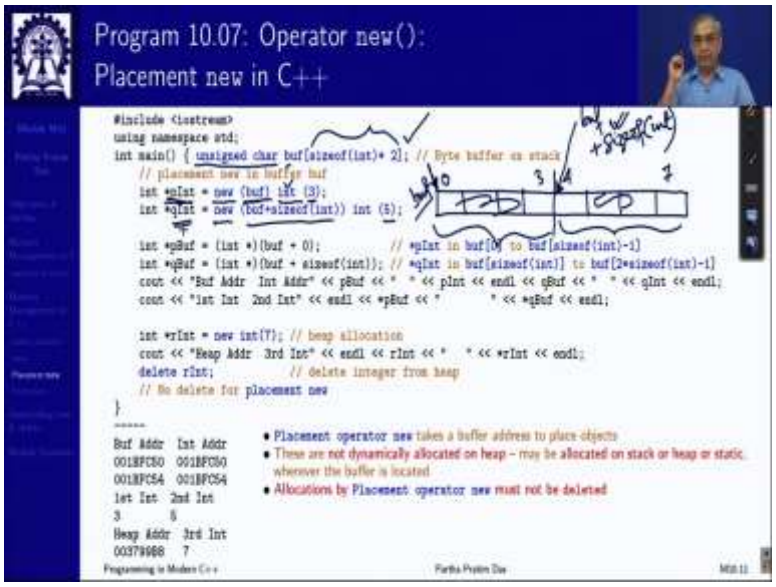
Here in C++ you have another operator called array new, so this operator is array new. It is operator new with square bracket. This part remains same the first parameter is a type, within square bracket the second parameter is the number of elements that the array will have. So this is something additional for C++, C did not give you any mechanism to specifically allocate memory for arrays.

It says that unless give you a chunk you decide whether it is a single element or multiple elements I have given you convertibility between pointer and array do whatever you want, but C++ one tells you exactly needs you to tell whether you want an isolated variable and or whether you want a array variable, and for the array you can pass the actual number of parameters, I am sorry, actual number of elements.

Similarly you have an operator delete, array delete now in which after delete you have to write this array symbol and then the array name and that will delete all the elements, release the entire array. It will do lot of other things, we will slowly come to that. So we have seen three things so far, one is operator new corresponding with operator delete.

One is function operator new corresponding with function operator delete that parallels malloc and free and one is array new corresponding to array delete which is specifically for dynamic allocation of arrays.

(Refer Slide Time: 14:27)



Program 10.07: Operator new(): Placement new in C++

```
#include <iostream>
using namespace std;
int main() { unsigned char buf[sizeof(int)* 2]; // Byte buffer on stack
    // placement new in buffer buf
    int *pInt = new (buf) int (3);
    int *qInt = new (buf+sizeof(int)) int (5);

    int *pBuf = (int *)(buf + 0);              // *pInt in buf[0] to buf[sizeof(int)-1]
    int *qBuf = (int *)(buf + sizeof(int)); // *qInt in buf[sizeof(int)] to buf[2*sizeof(int)-1]
    cout << "Buf Addr  Int Addr" << pBuf << "  " << pInt << endl << qBuf << "  " << qInt << endl;
    cout << "1st Int  2nd Int" << endl << *pBuf << "        " << *qBuf << endl;

    int *rInt = new int(7); // heap allocation
    cout << "Heap Addr  3rd Int" << endl << rInt << "   " << *rInt << endl;
    delete rInt;            // delete integer from heap
    // No delete for placement new
}
-----
Buf Addr  Int Addr
001BFC50  001BFC50
001BFC54  001BFC54
1st Int  2nd Int
3        5
Heap Addr  3rd Int
00379988   7
```

- Placement operator new takes a buffer address to place objects
- These are not dynamically allocated on heap – may be allocated on stack or heap or static, wherever the buffer is located
- Allocations by Placement operator new must not be deleted

So you can see that basically the dynamic memory management has been dealt with lot of importance in C++, because that is a key area of performance and quality of code that you can write. The story did not end here there is yet another operator that is called placement new, all earlier operator, operator new or operator new function or array new all of them dynamically creates the memory and gives you, so it is all in the free store.

Placement new is not that, placement new is where you provide the memory and it gives you the appropriate pointer from that memory. It is not dynamically created the user provides the memory and as if the operator places the object in the memory that you have given. The example will make it clear. Here is a memory chunk that I define.

What is that memory chunk? It is a byte array, unsigned character is always one byte so it is a byte array called buf and what is the size how many bytes will be there sizeof int times two so it is enough for two integers that is how I have created. So what I want I have this array and I want to dynamically create two integers one after the other in this buffer not create them on the free store, not in the heap.

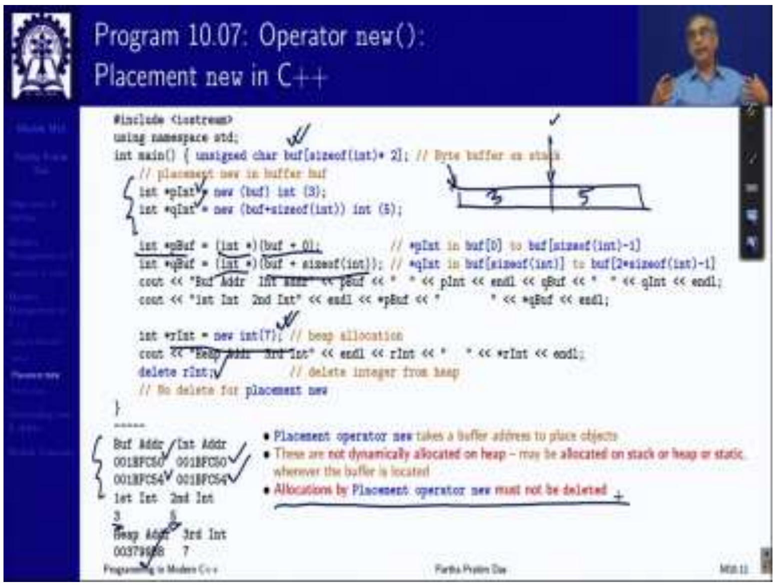
So the difference that I make is again new, it has to have the type, but before that within a pair of parentheses it takes a pointer to the memory where it will do this new creation and that is buf. So let me try to draw it here say assuming that int is of size four bytes so this is from zero to three is four bytes place enough for one integer and four to seven place enough for the second integer.

This is buf, so what I have given I have given buff, I have given this pointer to this placement new operator telling that give me an integer created at this location. Normal new would have got this address from the system by interacting with the free store, the memory manager of the system. Here I have given that address, so what happens is the pointer that I get back since I have given int it is a int* pointer has to be the same new behavior is same as this.

So my integer is here. The third parameter is the initialization value so over these four bytes the value three has been written in the integer format and I get that point. The second I want is I want the second integer to be created in the next four bytes, second integer to be created in the next four bytes so I again do new. This is my pointer now second integer has to come here how do I get that pointer?

This is buf and I have placed one integer so sizeof int, so this pointer this is buf plus sizeof int, buf plus four. So I pass that as a pointer where the integer should be created, I pass the initialization value five so over these four bytes, four to seven, five is created and I get back this pointer as q int. So it is all like new except the fact that I have provided the memory I have told how objects are to be placed.

(Refer Slide Time: 19:36)



Program 10.07: Operator new(): Placement new in C++

```
#include <iostream>
using namespace std;
int main() { unsigned char buf[sizeof(int)* 2]; // Byte buffer on stack
    // placement new in buffer buf
    int *pInt = new (buf) int (3);
    int *qInt = new (buf+sizeof(int)) int (5);

    int *pBuf = (int *)(buf + 0);              // *pInt in buf[0] to buf[sizeof(int)-1]
    int *qBuf = (int *)(buf + sizeof(int)); // *qInt in buf[sizeof(int)] to buf[2*sizeof(int)-1]
    cout << "Buf Addr  Int Addr" << pBuf << "  " << pInt << endl << qBuf << "  " << qInt << endl;
    cout << "1st Int  2nd Int" << endl << *pBuf << "       " << *qBuf << endl;

    int *rInt = new int(7); // heap allocation
    cout << "Heap Addr  3rd Int" << endl << rInt << "   " << *rInt << endl;
    delete rInt;            // delete integer from heap
    // No delete for placement new
}
-----
Buf Addr  Int Addr
001BFC50  001BFC50
001BFC54  001BFC54
1st Int  2nd Int
3        5
Heap Addr  3rd Int
003799B8   7
```

- Placement operator new takes a buffer address to place objects
- These are not dynamically allocated on heap – may be allocated on stack or heap or static, wherever the buffer is located
- Allocations by Placement operator new must not be deleted

Programming in Modern C++ Partha Pratim Das M10.11

There are deeper consequences of this which you will be able to understand once we have talked more about actual object creation and destruction and those processes then we will come back and revisit this again, but for now just understand this. What now if I try to look at these two addresses one here and one here the two integers as we have kept.

I have taken the buffer address by taking buf plus zero which is my first integer casting it to int*, I have taken buf plus size int this integer cast it to int* so these are the two direct buffer pointers and these are the two dynamic allocated integers in my address space and then I have printed the buffer address and this dynamic address to show that the buffer address and the integer address would be same in both cases which is what is expected.

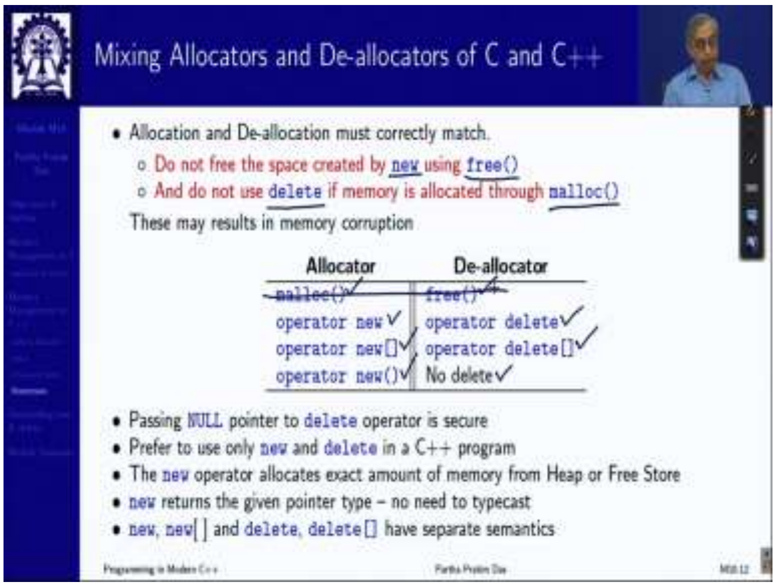
That is the behavior of the placement new and if you dereference these and print you will find the initialization values three and five. You must have got shaken, so just to clear your mind that this is not happening in the heap, in the free store do a normal new int seven. This new int seven is not placement it will do in the free store so you get a completely different type of address which is a free store address.

There is a major thing to remember, one is when you do new it is dynamically given from the system so you need to do a delete to release the memory but when you do placement new you have provided the memory, so the system does not know where this memory came from, like here I have given this buffer, which is a local variable of the function, which means that the buffer is on stack.

So this dynamically created object have actually been created on stack and when this function goes out of scope, when the function returns to its caller at that time the stack frame of this function containing this buffer will be deallocated and the memory will go. Right now you cannot remove this memory because it is a part of the stack frame which is active.

So, which means that allocations by placement operator new should never be deleted, that is one protocol you will have to remember. Now what is the specific use and all that, we will come slowly, just we are getting familiar with the different ways of handling dynamic allocation in C++ as compared to what we had in C.

(Refer Slide Time: 22:54)



Mixing Allocators and De-allocators of C and C++

- Allocation and De-allocation must correctly match.
  - Do not free the space created by new using free()
  - And do not use delete if memory is allocated through malloc()

  These may results in memory corruption

Allocator	De-allocator
~~malloc()~~ ✓	~~free()~~ ✓
operator new ✓	operator delete ✓
operator new[] ✓	operator delete[] ✓
operator new() ✓	No delete ✓

- Passing NULL pointer to delete operator is secure
- Prefer to use only new and delete in a C++ program
- The new operator allocates exact amount of memory from Heap or Free Store
- new returns the given pointer type – no need to typecast
- new, new[ ] and delete, delete[] have separate semantics

Programming in Modern C++ Partha Pratim Das M10.12

So now you have too much of allocators and deallocators because all C functions C dynamic memory management functions are in C standard library. So anything that is in C standard library is available in C++ you know, so you have all of them available so you have to have a very strict protocol of how you use them particularly in terms of allocation and reallocation.

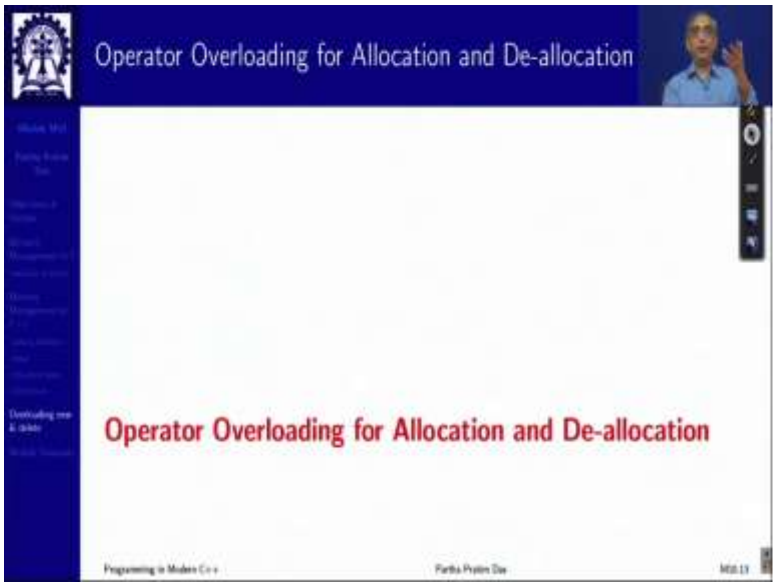
For example any allocation which is done by new must not be released by free or any allocation that is done by malloc must not be released by delete. They are completely different functions, their internal data structures are different, the management is different, and the consequence of this is unforeseen.

Since it is these are all run time calls compiler has no way to check that you are not freeing a memory by delete which was allocated by new, it is users responsibility to ensure that there is no shortcut on this. So always match your allocator with deallocator, if you have done malloc use free on that memory, if you have done operator new use operator delete that is one element allocation, deallocation.

If you have done array new, use array delete do not mix between them doing array new and doing operator delete not doing array deletes unpredictable results, if you have done placement new do not do a delete. Remember these things very clearly because many of you while developing programs with dynamic memory will often have these problems mixing and you will have get into difficulty.

I would also say that please make it a practice not to use this at all you do not need them. Just use the C++ ones, also note that null can be passed to the delete operator it is not an error and well everything else we have talked of here.

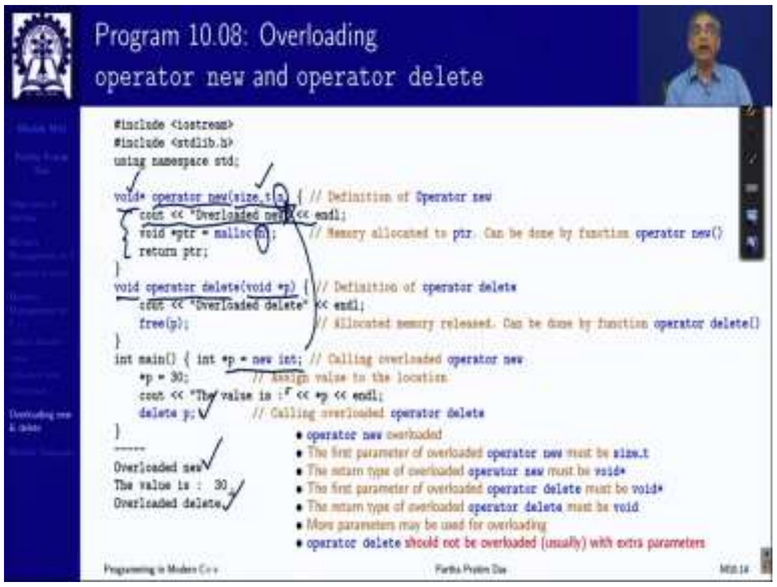
(Refer Slide Time: 25:41)



Operator Overloading for Allocation and De-allocation

Operator Overloading for Allocation and De-allocation

Programming in Modern C++ Partha Pratim Das M10.13



Program 10.08: Overloading operator new and operator delete

```
#include <iostream>
#include <stdlib.h>
using namespace std;

void* operator new(size_t n) { // Definition of Operator new
    cout << "Overloaded new" << endl;
    void *ptr = malloc(n);     // Memory allocated to ptr. Can be done by function operator new()
    return ptr;
}
void operator delete(void *p) { // Definition of operator delete
    cout << "Overloaded delete" << endl;
    free(p);                   // Allocated memory released. Can be done by function operator delete()
}
int main() { int *p = new int; // Calling overloaded operator new
    *p = 30;               // Assign value to the location
    cout << "The value is :" << *p << endl;
    delete p;              // Calling overloaded operator delete
}
-----
Overloaded new
The value is :  30
Overloaded delete
```

- operator new overloaded
- The first parameter of overloaded operator new must be size_t
- The return type of overloaded operator new must be void*
- The first parameter of overloaded operator delete must be void*
- The return type of overloaded operator delete must be void
- More parameters may be used for overloading
- operator delete should not be overloaded (usually) with extra parameters

Programming in Modern C++ Partha Pratim Das M10.14

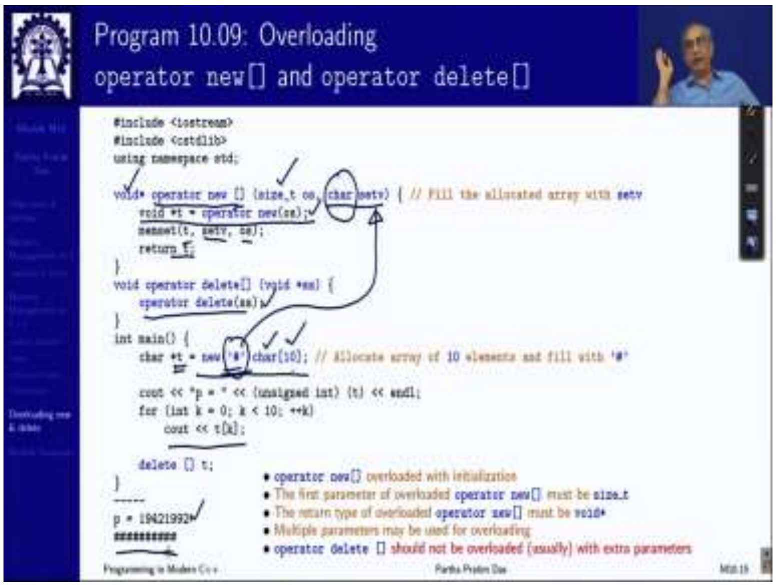
Now the last that I want to mention in terms of this is these are the operators as given but since these are operators they can be overloaded and it is okay to overload them for several reasons. So here is example of overloading operator new and operator delete. Now operator new must have this minimum signature, by minimum what I mean is its name is operator new, its first parameter will have to be a size t parameter in which the compiler actually passes the sizeof the type that it it is getting and its return type must be void*.

Again how that void* becomes the int* that you are getting is a is a mystery that we will solve later on, so with this you can, now I have written an allocation function here, I have done a malloc of n bytes, I could have used the operator new function also with n bytes and I have returned that void* pointer but in additionally I have just printed a message to show you that you are actually using your overloaded operator.

Similarly for operator delete the mandatory part of the signature is void* the first parameter which is the pointer that you are deleting and the return type has to be void you cannot change these things but you can add more parameters. Though adding more parameters to delete is not recommended, there is not much use for it but you can obviously add more parameters to operator new.

Now without that here now you do new int, this new int since it has been overloaded, this new int will now call your operator new. So the new int not only gets you the integer pointer, but it also prints the operator new message. Similarly when you do delete p it prints this overloaded delete message, overloaded new and overloaded delete messages. So that is proof that your overloaded operators are working, you can overload.

(Refer Slide Time: 28:25)

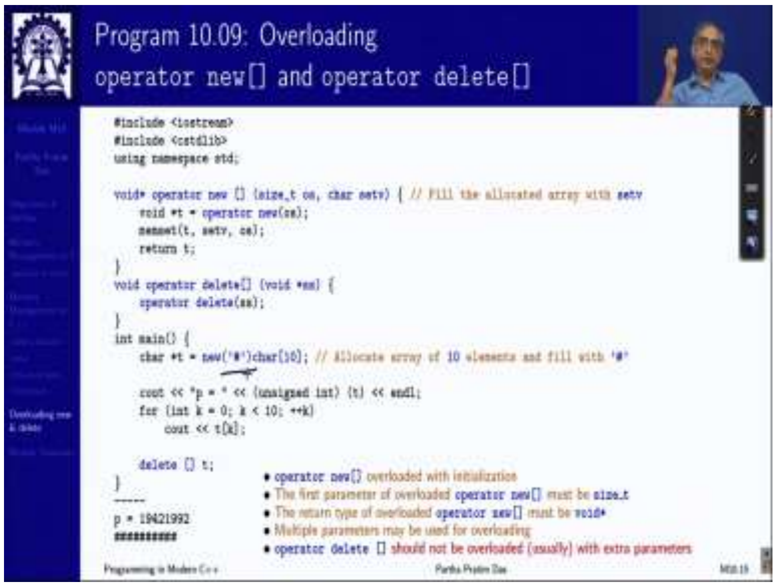


Program 10.09: Overloading operator new[] and operator delete[]

```
#include <iostream>
#include <cstdlib>
using namespace std;

void* operator new [] (size_t os, char setv) { // Fill the allocated array with setv
    void *t = operator new(os);
    memset(t, setv, os);
    return t;
}
void operator delete[] (void *ss) {
    operator delete(ss);
}
int main() {
    char *t = new('#')char[10]; // Allocate array of 10 elements and fill with '#'

    cout << "p = " << (unsigned int) (t) << endl;
    for (int k = 0; k < 10; ++k)
        cout << t[k];

    delete [] t;
}
-----
p = 19421992
##########
```

- operator new[] overloaded with initialization
- The first parameter of overloaded operator new[] must be size_t
- The return type of overloaded operator new[] must be void*
- Multiple parameters may be used for overloading
- operator delete [] should not be overloaded (usually) with extra parameters

Programming in Modern C++ Partha Pratim Das M10.15



Program 10.09: Overloading operator new[] and operator delete[]

```
#include <iostream>
#include <cstdlib>
using namespace std;

void* operator new [] (size_t os, char setv) { // Fill the allocated array with setv
    void *t = operator new(os);
    memset(t, setv, os);
    return t;
}
void operator delete[] (void *ss) {
    operator delete(ss);
}
int main() {
    char *t = new('#')char[10]; // Allocate array of 10 elements and fill with '#'

    cout << "p = " << (unsigned int) (t) << endl;
    for (int k = 0; k < 10; ++k)
        cout << t[k];

    delete [] t;
}
-----
p = 19421992
##########
```

- operator new[] overloaded with initialization
- The first parameter of overloaded operator new[] must be size_t
- The return type of overloaded operator new[] must be void*
- Multiple parameters may be used for overloading
- operator delete [] should not be overloaded (usually) with extra parameters

Programming in Modern C++ Partha Pratim Das M10.15

Similarly array new and array delete, so rules are the same, the operator name, operator new are assembled. The first operand is mandatory which is size_t, return type is void* just for illustration I have introduced another parameter, a second parameter. In the second parameter I am passing a character and what is the semantics, what I want? What I want is once the array is allocated it must be filled with that character all elements.

So what I do first is allocate using operator new function something which I did by malloc this is just for illustration you could have used malloc here also, get the pointer and then I am doing this to fill up with this character switch is one byte and naturally the array I have got is of bytes. So I do a memset, memset is a function available in in string.h, so which takes the pointer and the character and the size fills it up, sets into that simple way.

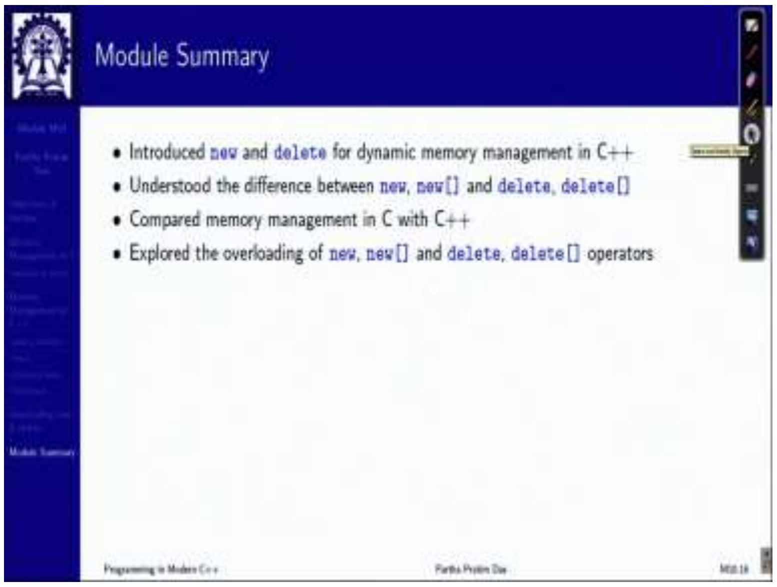
So everything has been filled by hash initialized and then I return t. This is different from the default behavior of array new operator; this fill up mechanism is not there. In terms of delete I do not do any additional smartness except that I am here I am releasing memory by operator delete function could have done free also, but try to use the matching one always.

So now when I invoke I pass the type, the number of elements and this syntax is little bit this way that way like the first parameter comes at the end, which is the sizeof number of elements and the second one comes at the beginning and type, sits in the middle but that is the way the syntax is worked out it is a little complicated syntax, but you will get used to it.

So here I am passing the character that I want as a filling character, so when I do this the t star array that I get and this is just to print the address of the array just to show you and then I write every element going through the loop and you can see that I am getting all these hashes, 10 hashes because I have initialized with that. So this kind of gives you an idea of what you can, I mean you can like this cat set v you can have more parameters also.

You can say my every even place should be one character my every odd place should be another character. You can do any kind of any number of parameters and so on; all that you will have to do here this got cluttered. Here within this parenthesis you have to put comma and provide these parameters, but you can get give your own meaning of dynamic allocation and deallocation for operator new as well as operator array new, you can do the same thing for placement operate also but you usually will not need that.

(Refer Slide Time: 32:35)



Module Summary

- Introduced `new` and `delete` for dynamic memory management in C++
- Understood the difference between `new`, `new[]` and `delete`, `delete[]`
- Compared memory management in C with C++
- Explored the overloading of `new`, `new[]` and `delete`, `delete[]` operators

Programming in Modern C++ Partha Pratim Das M10.16

So that is about the dynamic memory management here, so we have introduced new and delete for dynamic memory management in C++ and understood the difference between operator new and operator array new, operator delete and operator array delete, also the operator function placement new, and the basic protocol, the discipline of their use between C and C++, malloc with free, preferred you never should use, new with delete, operator array new with array delete, placement new not to be deleted.

So this is the basic way, there is a lot more of core object oriented part of C++ which will get integrated with this dynamic memory management subsequently, but this is with the built-in types with the types that you have in c you can use these operators in C++ and build your dynamic management completely in C++. Thank you very much for your attention and we will meet in the next module next week.