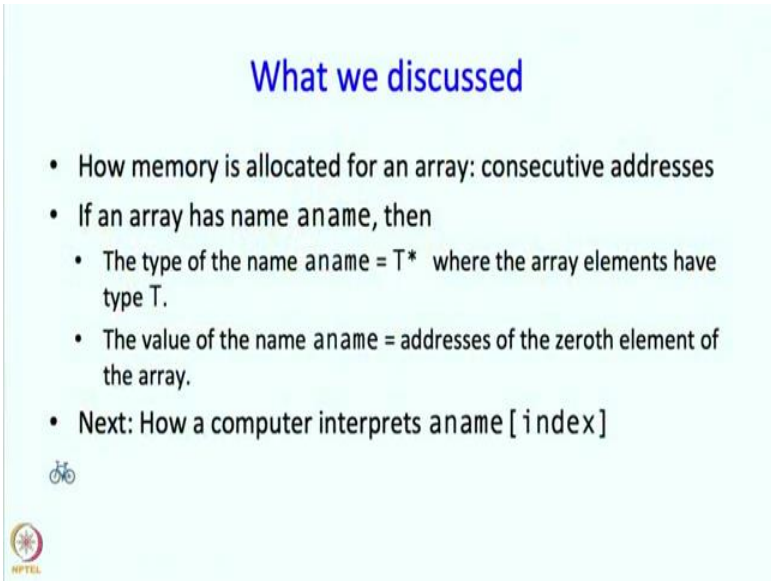
An Introduction to Programming through C++

Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture No. 16 Part - 2
Array Part – 2
Interpretation of aname [index]

(Refer Slide Time: 0:20)

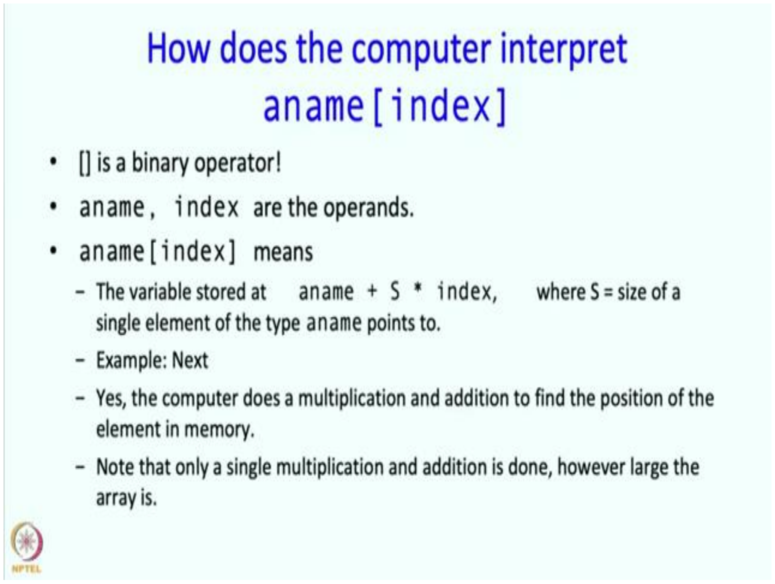


What we discussed

- How memory is allocated for an array: consecutive addresses
- If an array has name aname, then
  - The type of the name aname = T* where the array elements have type T.
  - The value of the name aname = addresses of the zeroth element of the array.
- Next: How a computer interprets aname[index]

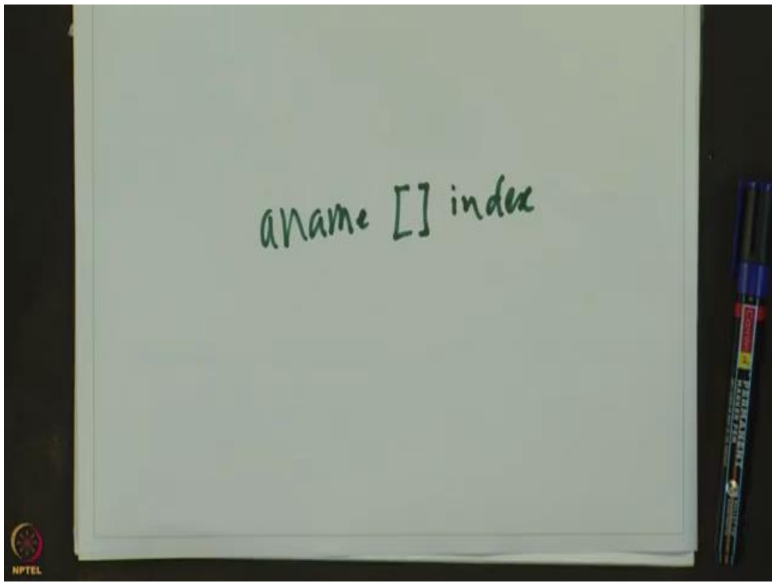
Welcome back, in the previous segment we discussed, how memory is allocated for an array. And we discussed what the value of, what value the name of an array has and we also discussed what type the name of an array has.

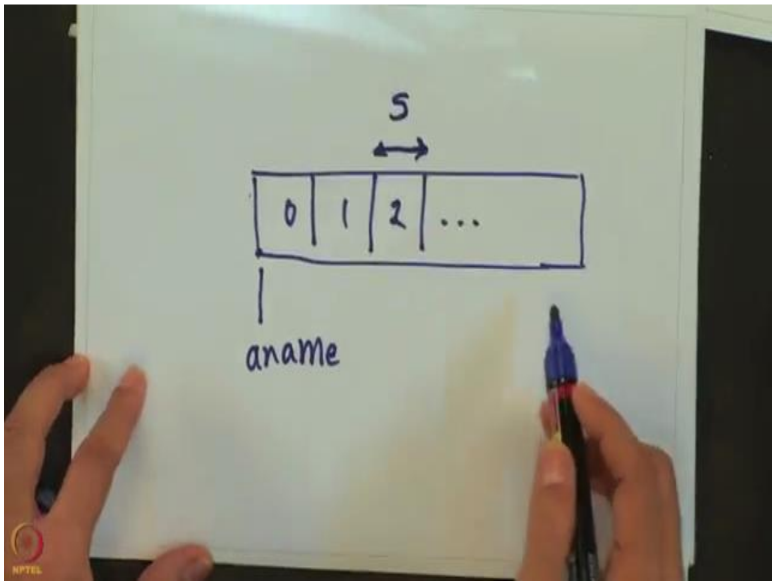
(Refer Slide Time: 0:45)



How does the computer interpret aname[index]

- [] is a binary operator!
- `aname, index` are the operands.
- `aname[index]` means
  - The variable stored at `aname + S * index`, where S = size of a single element of the type `aname` points to.
  - Example: Next
  - Yes, the computer does a multiplication and addition to find the position of the element in memory.
  - Note that only a single multiplication and addition is done, however large the array is.





In this segment we are going to discuss, discuss how a computer interprets something like aname[index]. So the first observation or the first thing that I should tell you is that square brackets are actually considered to be a binary operator by C++. So we could have written this expression as aname[]index, this may look, make it look more like an operator expression, but because we are familiar with putting things inside brackets we choose to write it as aname open bracket, index, close bracket. But really the interpretation of C++ is sort of like this.

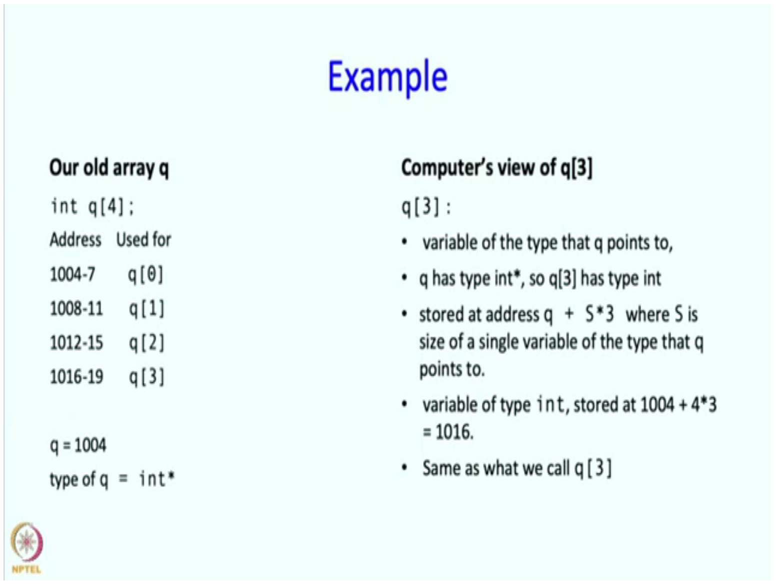
Alright, so what exactly is the interpretation? Well so first of all aname and index are the operands of the operator and aname of index means the following. So it means, this expression means a variable. And which variable? The variable which is stored at aname plus S times index, where S is the size of a single element of the type that aname points to. So this is a little bit of complicated definition and I am going to give an example which will make it clear.

And you may say that look when I am doing this interpreting this expression or whenever I write, whenever I write in my program aname[index], does the computer actually compute something like this? Yes, it does. So it will multiply the index by something and then add it to the name. So this is not really entirely an, entirely surprising, because we said that C++ is going to give you a region of memory and the region is going to start at aname and then you are going to have element 0, then you are going to have element 1, you are going to have element 2 and so on.

And this distance is going to be the size of each element or it is this S. And therefore, if I want to get to the ith element I should know how much forward I should go and how much forward should I go? I should go forward S times whatever that index is, that is it. So that is really this calculation.

Alright, so you have to note that a single multiplication and addition is done, no matter how large the array is. So this is why an array is often called a random access data structure, so you can get to any element of that array in essentially the same amount of time by doing just a single multiplication and addition.

(Refer Slide Time: 4:03)



Example

Our old array q

`int q[4];`

Address	Used for
1004-7	`q[0]`
1008-11	`q[1]`
1012-15	`q[2]`
1016-19	`q[3]`

q = 1004

type of q = `int*`

Computer's view of q[3]

`q[3]`:

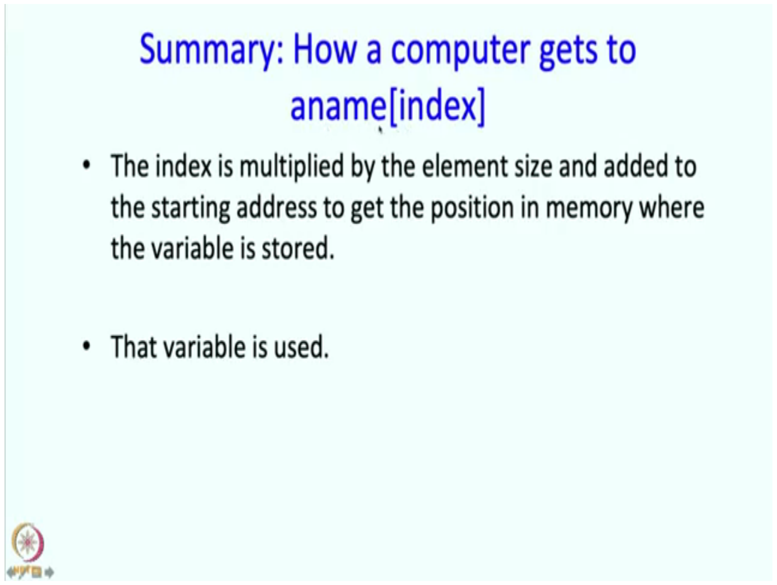
- variable of the type that q points to,
- q has type int*, so q[3] has type int
- stored at address `q + S*3` where S is size of a single variable of the type that q points to.
- variable of type `int`, stored at 1004 + 4*3 = 1016.
- Same as what we call `q[3]`

So here is an example, so this is our old array q, so we defined it by writing int q 4 and we said earlier that these are the addresses used for q0, q1, q2, q3 and we said that the name q has value 4 and its type is int*. So let us take one of the elements, say q3 and let us see how the computer or how C++ views q3 and whether that view matches our view. So what is q3? So we said that first of all it is a variable, so q this indexing operator 3 is an expression and the result of the expression is, it is variable of the type that q points to.

And what does q point to? q is int*, so q points to an int, so q3 is an int which is good because we are expecting q3 to be an int. And q has type int*, so q3 has type int. And it is stored at address q plus S times 3 where S is the size of a single variable of the type that q points to. In other words, it is a variable of type int which is what we already said and it is stored at q which is 1004, S which is 4 times 3 so 1016, so it is the starting address so it is q3.

So it is indeed what we think of q3 as what we think of as q3 so the computers view and our view actually do match. And it is not a surprise, because I told you more pictorially how the computer gets to the position in memory where q3 is or which element of which variable in memory, so it needs to get to that position in memory and that calculation clearly should require addition and multiplication.

(Refer Slide Time: 6:15)

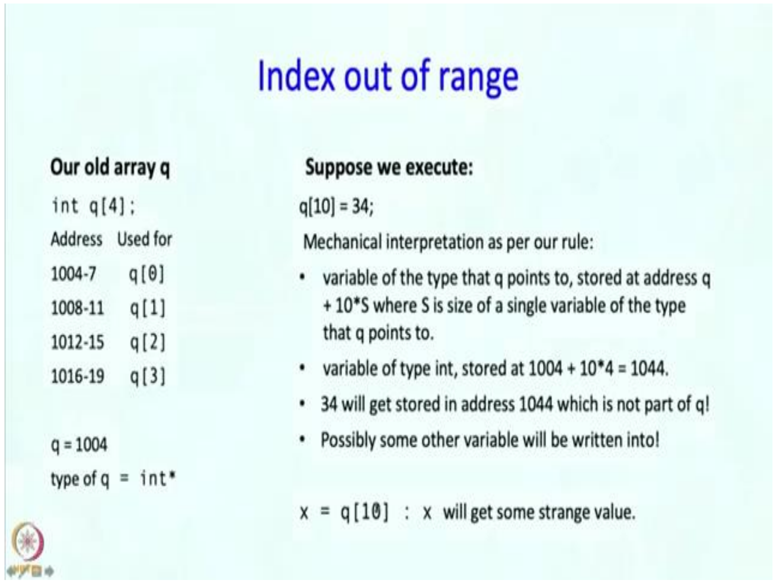


Summary: How a computer gets to aname[index]

- The index is multiplied by the element size and added to the starting address to get the position in memory where the variable is stored.
- That variable is used.

So, how a computer gets to aname and address? The index is multiplied by the element size and added to the starting address to get the position in memory where the variable is stored. And whenever you write aname[index] that variable which you got to by doing this address calculation is the one which is considered the result of this expression, the result of evaluating this expression.

(Refer Slide Time: 6:46)



Index out of range

Our old array q

```
int q[4];
```

Address	Used for
1004-7	q[0]
1008-11	q[1]
1012-15	q[2]
1016-19	q[3]

q = 1004

type of q = int*

Suppose we execute:

q[10] = 34;

Mechanical interpretation as per our rule:

- variable of the type that q points to, stored at address q + 10*S where S is size of a single variable of the type that q points to.
- variable of type int, stored at 1004 + 10*4 = 1044.
- 34 will get stored in address 1044 which is not part of q!
- Possibly some other variable will be written into!

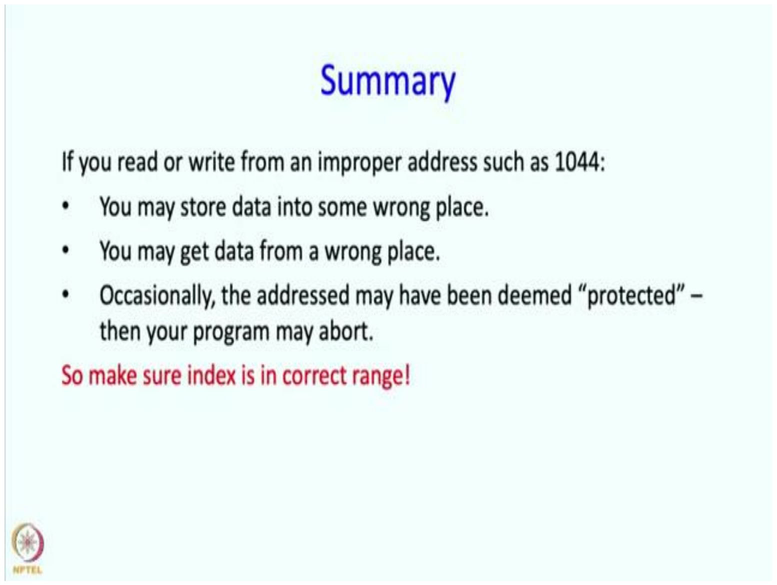
x = q[10] : x will get some strange value.

Now, in light of this discussion of how aname[index] works, let us try to see what index out of range means. So here is our old array q and suppose we execute q[10]=34, so what would this do? So if we mechanically interpret as per our rule and, which is how the computer interprets it. So it is the variable of type that q points to, stored at address q plus 10 times S where S is the size of an integer of a single variable of the type that q points to. Again this is a mouthful, but we know how to interpret it now.

So it is a variable of type int, because q points to int, so it is a variable of type int. It is stored at q plus 10 times S so q plus 10 times 4 equal to 1044, so this is the position where q of 10 would have been had q actually had 10 elements, but q does not have 10 elements. So this address is somewhere beyond the region allocated for q, so 34 if you execute this statement 34 will get stored in some strange part of the memory which has nothing to do with q. So that is why it is bad to have indices out of range, so some other variable will get destroyed.

Now the other way is also bad, if I write x equals q of 10 what will happen? It will pick up the value from this 1044 and so X will get some strange value, who knows what value it gets? So again it is a, it is a bad idea to have an address which is out of range. And now you see, what C++ will do in these cases, in in such cases.

(Refer Slide Time: 9:05)



Summary

If you read or write from an improper address such as 1044:

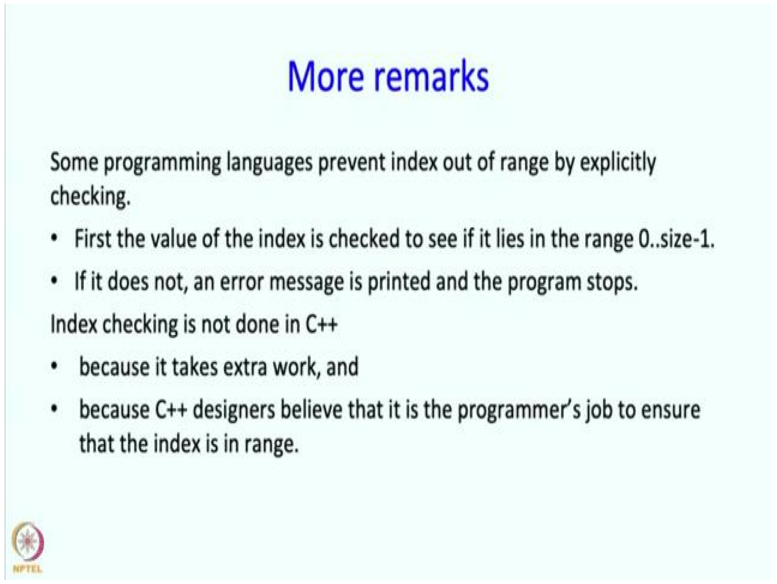
- You may store data into some wrong place.
- You may get data from a wrong place.
- Occasionally, the addressed may have been deemed "protected" – then your program may abort.

So make sure index is in correct range!

But there is a little bit more to be said, if you read or write from an improper address such as 1044, you may store data into wrong, into some wrong place, you may read data from a wrong place. And sometimes, C++, or your computer hardware may say that look some addresses are protected, why are they protected? Because they may contain code and you do not want programs to be writing data into the code region of memory or the program region of memory.

So, if your program tries to do that then the hardware will raise an alarm and it will cause your program to abort. So any of these things could happen and so therefore make sure that the index is in the correct range.

(Refer Slide Time: 9:58)



More remarks

Some programming languages prevent index out of range by explicitly checking.

- First the value of the index is checked to see if it lies in the range 0..size-1.
- If it does not, an error message is printed and the program stops.

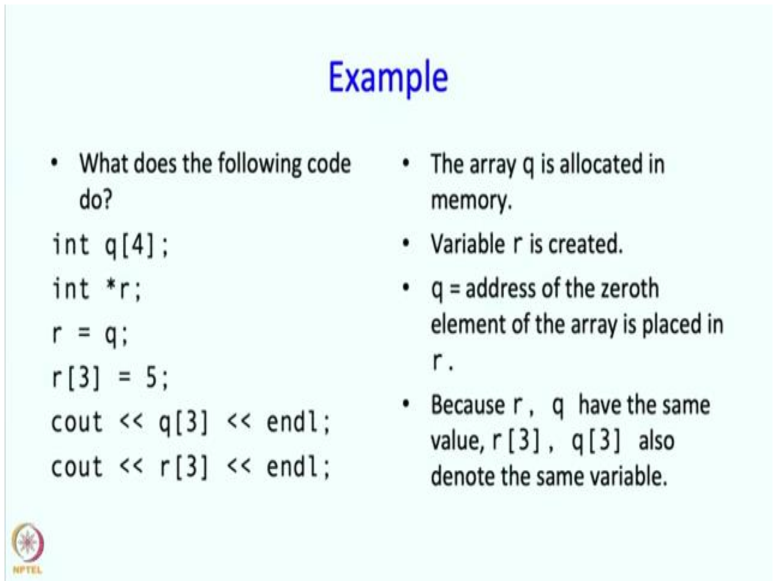
Index checking is not done in C++

- because it takes extra work, and
- because C++ designers believe that it is the programmer's job to ensure that the index is in range.

Now some programming languages prevent index out of range by explicitly checking, so the moment you write a[i] the language will have additional code which it generates itself you do not have to write it, which will check first whether i lies in the range 0 through size minus 1, where this size is the size of the array in question. If it does not, then it will not make that access, it will just print an error message saying that oh you made, you gave an index out of range and the program will stop.

Index checking is not done in C++, why? Because it takes extra work, C++ wants to run very fast. Of course C++ does not want wrong answers, but C++ says, the C++ designers believe that it is the programmer's job you have to ensure that your index is correct and of course you can do that, I mean you have to think a little and you to be careful but that is your job, that is what the C++ designers believe.

(Refer Slide Time: 11:22)



Example

- What does the following code do?

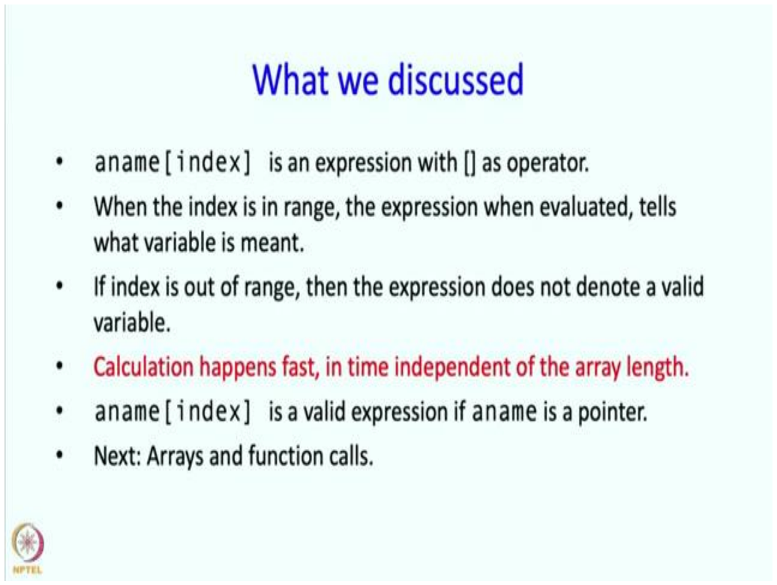
```
int q[4];
int *r;
r = q;
r[3] = 5;
cout << q[3] << endl;
cout << r[3] << endl;
```

- The array q is allocated in memory.
- Variable r is created.
- q = address of the zeroth element of the array is placed in r.
- Because r, q have the same value, r[3], q[3] also denote the same variable.

So here is an example, so we have this code and let us see what it does. So when you come to the first statement of this code, this statement will cause an array q to be allocated memory, what happens next? The variable r is created, remember the type of r is int*, so it is meant for storing addresses so when you execute this r will get the value q, what is the value of q? It is the starting address, so what will the value of r be? It will also be the starting address, so at this point r and q will have the same value.

Alright, so now what happens when you execute this? When you execute this, this expression since r has the same value as q, this expression is as good as q[3]. So 5 will get placed in q[3], so when you do this printing 5 in get printed. And if you print r[3], this is not, this is also going to refer to q[3] and so again 5 will get printed. So r and q have the same value and r[3] and q[3] denote the same variable and therefore this will end up printing 5 in both cases.

(Refer Slide Time: 12:58)



What we discussed

- `aname[index]` is an expression with [] as operator.
- When the index is in range, the expression when evaluated, tells what variable is meant.
- If index is out of range, then the expression does not denote a valid variable.
- Calculation happens fast, in time independent of the array length.
- `aname[index]` is a valid expression if `aname` is a pointer.
- Next: Arrays and function calls.

Aright, so what have we discussed? So we have discussed that aname[index] is an expression with square bracket as an operator. When the index is in the range the expression evaluate, when evaluated tells what variable is meant, if the index is out of the range, then the expression does not denote a valid variable. And this calculation happens fast, there is only an addition and multiplication to be done and it happens in time independent of the array length. Aname[index] is a valid expression if aname is a pointer.

Next all these things are going to have some bearing on how you use arrays in function calls and that is what we will see in the next segment, we will take a break.