



**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- 3**  
**Array Part-2**  
**Arrays and function calls**

Welcome back, in the previous segment we discussed what how `aname[index]` is interpreted.

(Refer Slide Time: 0:24)

What we discussed

- `aname [ i ndex ]` 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 [ i ndex ]` is a valid expression if `aname` is a pointer.
- Next: Arrays and function calls.


Basically we discussed the semantics of this square bracket operator. In this next segment we are going to talk about how arrays are passed to functions.

(Refer Slide Time: 0:43)

### Function calls on arrays

We might like to write functions to:

- find the largest value in the array
- find whether a given value is present in the array.



So we really would like to have functions on arrays, right. I mean we might do, we might want to do the following things, we might say want a function which finds the largest value in an array.



(Refer Slide Time: 0:52)

### Function calls on array

We might like to write functions to:

- find the largest value in the array
- find whether a given value is present in the array.
- find the average of the elements in the array.
- ...

We see how to write functions involving arrays next.




Or if you might want a function which decides whether a given value is present in an array or a function which calculates the average of the elements in array, in an array. And so many other things could be done, okay. So in this segment we are going to see how all of these can be done, all of these things can be done.

(Refer Slide Time: 1:13)

### A program to find the average of elements in array

```
double avg(double* M, int
n){
    double sum = 0;
    for(int i=0; i<n; i++)
        sum +=M[i];
    return sum/n;
}
int main(){
    double q[]={11,12,13,14};
    cout << avg(q, 4) << endl;
}
```




Let me just plunge into it directly by giving an example. So here is a program which finds the average of elements in an array or a function rather. So this is the function and this is the main program which calls this function, okay. So before looking at it too carefully and trying to see how it executes,

(Refer Slide Time: 1:42)

### A program to find the average of elements in array

```
double avg(double* M, int
n){
    double sum = 0;
    for(int i=0; i<n; i++)
        sum +=M[i];
    return sum/n;
}
int main(){
    double q[]={11,12,13,14};
    cout << avg(q, 4) << endl;
}
```

- Let us first check if this is a syntactically valid program, never mind what it does.
- The types of **the arguments to a call** must match the types of **the parameters**.
- The first parameter of avg has type double\*
- The first argument in the call is q, whose type is double\*, because it points to the first element of a double array.
- The second parameter is of type int, and 4 in the call is indeed an int.



let us first make some preliminary checks, okay. Let us first check if this is a syntactically valid program, never mind what it does, okay. So what do I mean by that? Well, when we make a function call, when we, okay, so, the types of the arguments must match the types of the parameters to the function, okay, so is that happening here. Well the first parameter to average has type double star, okay. So this is the first parameter and its type is double\*, okay.

The first argument in the call is `q` and its type if you remember is also a `double*`, why? Because it's the name of an array of doubles and its value is the address of the zeroth element and it points to a double element. So the type is correct, the type of this is indeed the same as the type of this. So when we execute the function, the value `q` will indeed be copied into `M`, so no problem with that.

The second parameter here is type `int` and here also is type `int`, so that is of course absolutely no problem about that, so that is nothing new about that.

(Refer Slide Time: 3:09)

### A program to find the average of elements in array

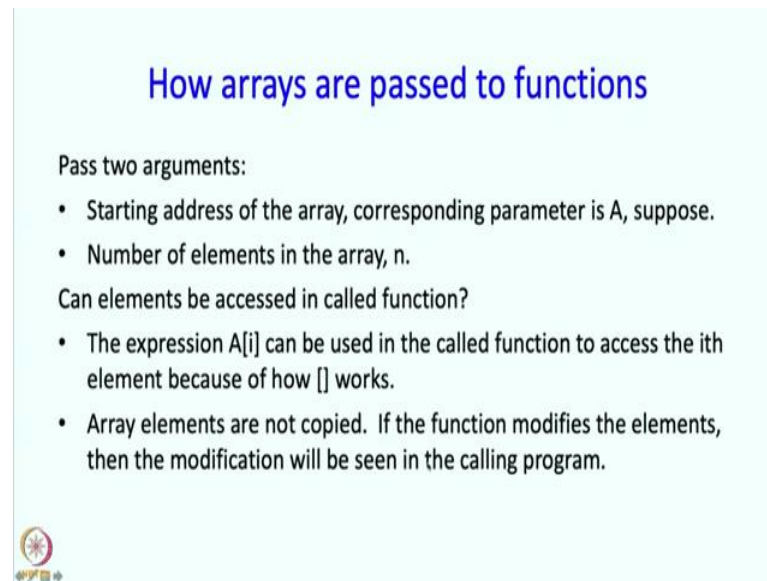
|  |   |
|--|---|
| <pre>double avg(double* M, int n){     double sum = 0;     for(int i=0; i&lt;n; i++)         sum +=M[i];     return sum/n; }  int main(){     double q[]={11,12,13,14};     cout &lt;&lt; avg(q, 4) &lt;&lt; endl; }</pre> | <p>On execution of <code>avg(q, 4)</code> in main</p> <ul style="list-style-type: none"> <li>Activation frame created for <code>avg</code>.</li> <li>Value of <code>q</code> (starting address of array) copied into parameter <code>M</code>.</li> <li>In each iteration, <code>M[i]</code> is needed.</li> <li><code>M</code> has value = starting address of <code>q</code>, and type <code>double*</code></li> <li><code>M[i]</code> means "Variable of type <code>double</code> at address <code>q + 4*i</code>"</li> <li>Thus <code>M[i]</code> in <code>avg</code> means <code>q[i]</code> of main.</li> <li>Thus average of the elements of <code>q</code> is calculated in <code>avg</code>.</li> <li>The average is returned, and printed.</li> </ul> |
|--|---|

Okay, now let us see how this is going to be executed. So when we execute this average of `q`, 4 what is going to happen? Well activation frame will be created for average and the value of `q` which is the starting address of this array will get copied into the parameter `M`. And in each iteration what is going to happen? `M[i]` is needed. Now like the example that we took a little bit earlier, this `M` has exactly the same value as this `q`. So if we write `M[i]` in this, even in this, `M[i]` is going to be in the same variable as `q[i]`. Because `M` and `q` mean the same thing, `M` and `q` have the same value, okay. So `M` has the value starting address of `q` of type `double`, so `M[i]` means variable of type `double` at address `q` plus `8i`.

Or otherwise it is the same thing as `q[i]`. Exactly like the exercise that we just, example that we justed in the previous segment. So effectively what is going to happen is that we are going to calculate the average of `q0`, `q1`, `q2`, `q3`, which is exactly what we want. And so this will indeed return, it will calculate the sum of the elements and then return divided by four, so it will actually return the average exactly like we wanted.

And the main program will print it out, so that is how the execution will go.

(Refer Slide Time: 5:03)



### How arrays are passed to functions

Pass two arguments:

- Starting address of the array, corresponding parameter is A, suppose.
- Number of elements in the array, n.

Can elements be accessed in called function?


- The expression `A[i]` can be used in the called function to access the *i*th element because of how `[]` works.
- Array elements are not copied. If the function modifies the elements, then the modification will be seen in the calling program.

So how are arrays then passed to functions from this example? So we are passing two arguments, the starting address of the array and let say the corresponding parameter is A, okay, and the number of elements in the array. Or maybe I should have written M to keep consistency with the previous example, but let us say it is A. So now can elements of the array be accessed in the called function? Well, if you write `A[i]` in the called function, it will actually give you the *i*th element of the calling program. And therefore, you can access the elements, even though you are not, you are not copying the elements, you are just sending the starting address of that array. But if the function modifies the elements, then the modification will be seen in the main program because the function is directly operating with the same array. It has a pointer and that pointer is operating directly in the activation frame of the calling program.

(Refer Slide Time: 6:32)

### Remarks

- When you pass the name of an array in a function call, its value, the starting address of the array, is copied to the corresponding parameter.
- But because we pass the starting address of the array, we are effectively enabling the function to pass the array.
- Can say
  - "Array name is passed by value"



So when you pass the name of an array in a function call, its value the starting address of the array, is copied to the corresponding parameter. But because we pass the starting address of the array, we are effectively enabling the function to pass the array. You can say the following about this entire process. You can say that the array name is passed by value. So let us just, let me just explain this.

(Refer Slide Time: 7:06)


### A program to find the average of elements in array

```
double avg(double* M, int n){
    double sum = 0;
    for(int i=0; i<n; i++)
        sum +=M[i];
    return sum/n;
}

int main(){
    double q[]={11,12,13,14};
    cout << avg(q,4) << endl;
}
```

On execution of avg(q, 4) in main

- Activation frame created for avg.
- Value of q (starting address of array) copied into parameter M.
- In each iteration, M[i] is needed.
- M has value = starting address of q, and type double\*
- M[i] means "Variable of type double at address q + 4\*i"
- Thus M[i] in avg means q[i] of main.
- Thus average of the elements of q is calculated in avg.
- The average is returned, and printed.



## How arrays are passed to functions

Pass two arguments:

- Starting address of the array, corresponding parameter is A, suppose.
- Number of elements in the array, n.

Can elements be accessed in called function?

- The expression `A[i]` can be used in the called function to access the *i*th element because of how `[]` works.
- Array elements are not copied. If the function modifies the elements, then the modification will be seen in the calling program.



## Remarks

- When you pass the name of an array in a function call, its value, the starting address of the array, is copied to the corresponding parameter.
- But because we pass the starting address of the array, we are effectively enabling the function to pass the array.
- Can say
  - "Array name is passed by value"



So in this, this array name is passed by value, why? Because the value of `q` is put in the value of, put in `M`. And so therefore, `M` and `q` have the same value. But indirectly, indirectly because you are passing the array name, you are allowing the called function access to the entire array. So you could say that the array elements are passed by pointer. Well effectively, you are saying that all the array elements are passed by a pointer to the zeroth element, because if you add the appropriate displacement, the called function can get to every element which it did in our `average` function and it can do in any old function.

The interesting part in all of this is the square bracket operator. So it given the address of any array and an index, it can get us to the corresponding element. Even if the address belongs to a different activation frame.

(Refer Slide Time: 8:23)



## Remarks

- An alternate syntax is also allowed.  
`int avg(double M[], int n){...}`
- In this `double M[]` is synonymous with `double* M`, but slightly more indicative that we expect `M` to be an array name, and not just any old pointer.
- The second argument to `avg` is not “required” to be the array length. If it is smaller, then the function will return the average of just that part of the array.

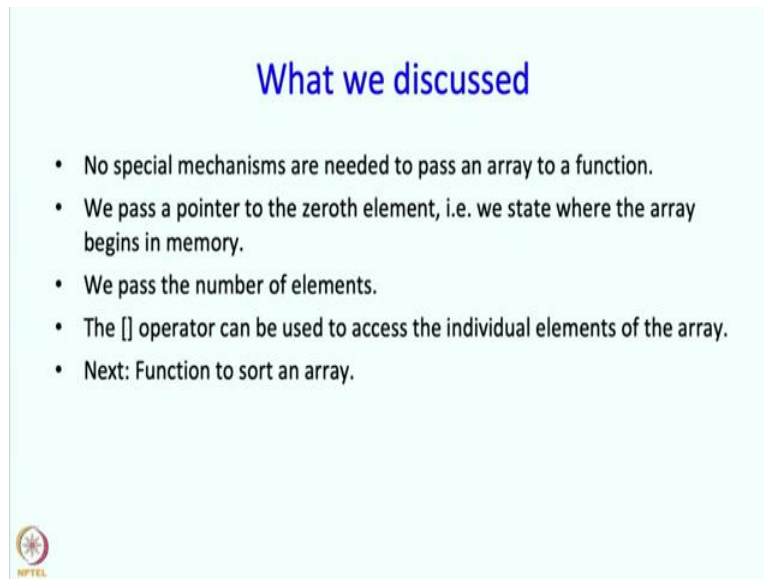


Now when we write functions an alternate syntax is also allowed and maybe it is recommended. So I can write the average function as not `double* M`, but `double M[]`. In this `double M[]` is synonymous with `double *M`, okay, it means the same thing. But it is slightly more indicative that we expect `M` to be an array name and not just any old pointer. But this is really for our benefit, C++ treats both of these things as essentially the same.

And I should further note that this `M[]` syntax is not allowed in other places besides the declaration or definition of a function. Now the second argument over here okay, is not really linked formally to this, this element. It is just a second argument, it is just another integer. What you do with that integer inside the function body, is your problem. So we are choosing, we are deciding when we write this function that the second argument is supposed to be the length of the array. So, of course, the average function does not know that. So you can in fact pass a smaller value of `n` and then the average function will just take the average of the initial so many elements, rather than all the elements in the array. All right, so what have we discussed?




(Refer Slide Time: 10:20)



### What we discussed

- No special mechanisms are needed to pass an array to a function.
- We pass a pointer to the zeroth element, i.e. we state where the array begins in memory.
- We pass the number of elements.
- The [] operator can be used to access the individual elements of the array.
- Next: Function to sort an array.



So we have said that, no special mechanisms are needed to pass an array to a function other than the semantics of the square bracket operator. And also the idea that the array name is a pointer to the starting address, the starting address of the region allocated for the array. We pass a pointer to the zeroth element that is we state where the array begins in memory and we pass the number of elements. This is what we do if we want to pass an array to a function.

The square operator can be used to access individual elements of the array. And this concludes this segment. In the next segment, we are going to build a slightly more elaborate function, a function to sort an array, but will take a quick break.