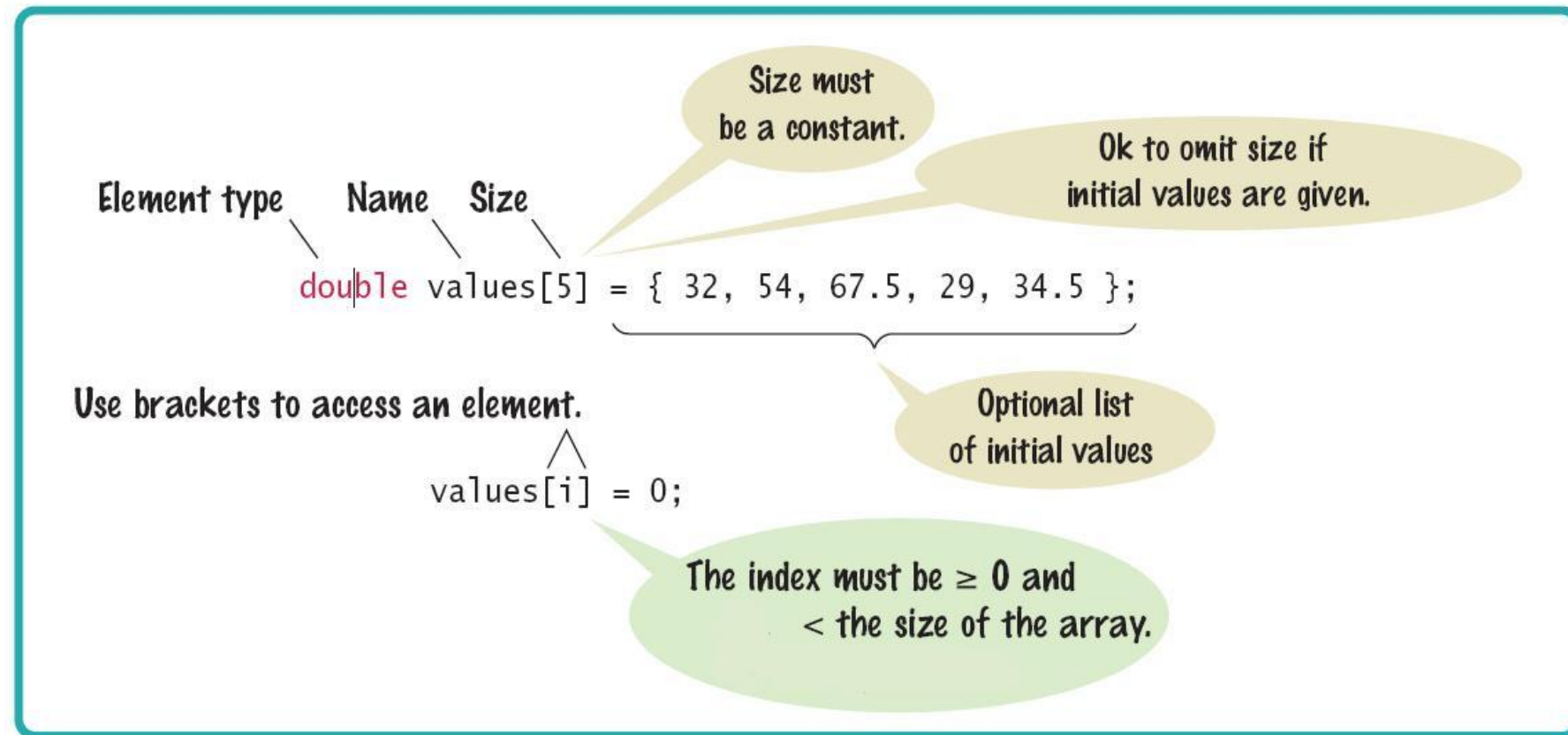


More Arrays

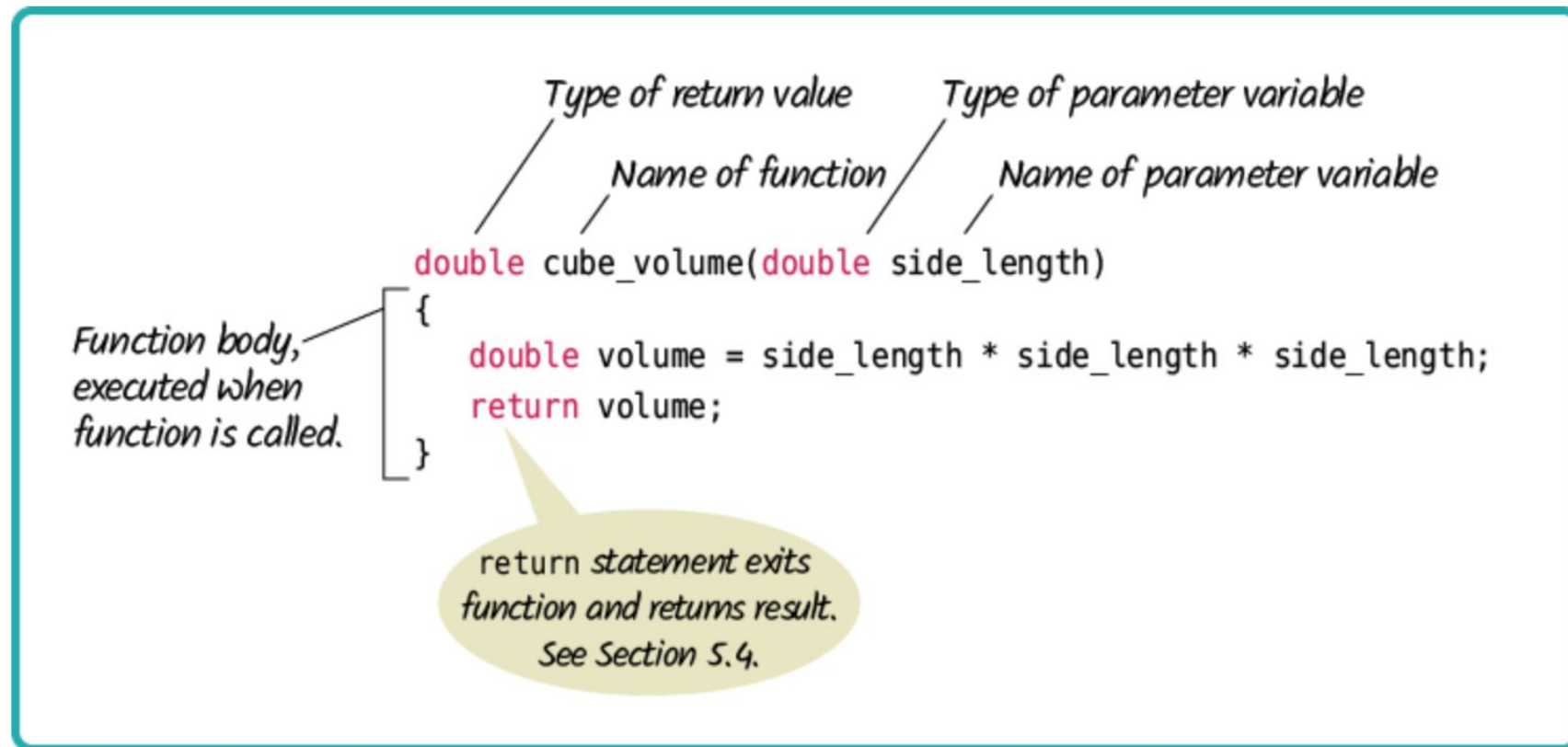
Recap: Array

Defining an Array



Recap: Functions

Syntax 5.1 Function Definition



Commonly used Array Algorithms

Insertion

- Let's suppose we are given an array of elements (consider integers for simplicity)
- We can insert an element in two ways
 - One at the end of the array (if there is sufficient space, ofcourse!)
 - On the other hand, anywhere in the array

0	1	2	3	4	5	6	7	8	9
1	3	5	7	9	11				

Insertion - Scenario 1

0	1	2	3	4	5	6	7	8	9
1	3	5	7	9	11				

- Suppose, we wish to insert 13 at the end of the array.
- What do you need?
 - The position where you need to insert
 - And the element to insert, indeed
 - Simply equate array at the index, to the element
- Let's remember to do some validation too!

Insertion - Scenario 2

0	1	2	3	4	5	6	7	8	9
1	3	5	7	9	11	13			

- Suppose, we wish to insert 1 at the index #1
- What do you need?
 - We know the element and the position.
 - But how to insert?



Insertion - Scenario 2

0	1	2	3	4	5	6	7	8	9
1	3	5	7	9	11	13			

- Suppose, we wish to insert 1 at the index #1
- To insert,
 - Make space for this new element
 - Move everything from the position one step to the right
 - Add the element at the position



Deletion

- Similarly, we can delete an element in two ways
 - Delete an element given it's index
 - Delete the element at index 6
 - Delete an element, given it's value
 - Delete the element 7 from the array

0	1	2	3	4	5	6	7	8	9
1	1	2	3	5	7	8	11	13	

Deletion - Scenario 2

0	1	2	3	4	5	6	7	8	9
1	1	2	3	5	7	8	11	13	

- Suppose, we wish to delete element 7.
 - Let's first find it's index.
 - Move everything after it, one position to the left



Arrays as Parameters in Functions

Pass by value

```
double cube_volume(double side_length)
{
    if (side_length >= 0)
    {
        return side_length * side_length * side_length;
    }
}

int main()
{
    int len = 3;
    double result1 = cube_volume(len); // Use of cube_volume
    cout << "A cube with side length 2 has volume " << result1 << endl;
    return 0;
}
```

Arrays as Parameters in Functions

- Recall that when we work with arrays we use a companion variable.
- The same concept applies when using arrays as parameters:
- You must pass the size to the function so it will know how many elements to work with.

Array as function argument

- What does the computer know about an array?
 - The base type
 - The address of the first indexed variable
 - The number of indexed variables
- What does a function know about an array argument?
 - The base type
 - The address of the first indexed variable

Entire Arrays as Arguments

- Formal input parameter argument can be an entire array!
 - argument passed in function using array name
 - called array parameter
- Send size of array as well
 - typically done as second parameter
 - simple int type formal parameter

Arrays as function argument

In some main() function definition, consider this call:

```
int score[5], numberOfScores = 5;  
fillup(score, numberOfScores);
```

- 1st argument is entire array
- 2nd argument is integer value
- **No brackets on the array argument**
- Passing in score → provides fillup() with the data type (int) and address of score[0]
 - knowing the type helps us retrieve the 2nd-last elements
- Passing in numberOfScores → provides size of array

Arrays as function argument

In some main() function definition, consider this call:

```
int score[5], numberOfScores = 5;  
fillup(score, numberOfScores);
```

- Must send in size of array separately
- **Fun fact:** This means we can use the **same function** to fill any sized array!
 - Exemplifies nice “re-use” properties of functions

```
int score[5], time[10];  
fillup(score, 5);  
fillup(time, 10);
```

Array as function argument: How?

- What's really passed?
- Think of array as 3 "pieces"
 - Address of first indexed variable (`arrName[0]`)
 - Array base type (`int` or `double` or `float` or ...)
 - Size of array
- Only 1st piece is passed!
 - Just the beginning address of array (the 1st element)
 - Knowing the type helps us retrieve the (2nd – last) elements

Array Parameters in Functions Require [] in the Header

- You use an empty pair of square brackets *after* the parameter variable's name to indicate you are passing an array.

```
double sum(double data[], int size)
```

Array Function Call Does NOT Use the Brackets

When you call the function, supply both the name of the array and the size, BUT NO SQUARE BRACKETS!!

```
double NUMBER_OF_SCORES = 10;
double scores[NUMBER_OF_SCORES] = { 32, 54, 67.5, 29, 34.5,
80, 115, 44.5, 100, 65 };
double total_score = sum(scores, NUMBER_OF_SCORES);
```

You can also pass a smaller size to the function:

```
double partial_score = sum(scores, 5);
```

This will sum over only the first five doubles in the array.

Arrays as Parameters but No Array Returns

- You can pass an array into a function but you cannot return an array.
- However, the function can modify an input array, so the function definition must include the result array in the parentheses if one is desired.

Array Parameters Always are Reference Parameters

- When you pass an array into a function, the contents of the array can *always* be changed. An array name is actually a reference, that is, a memory address:

```
//function to scale all elements in array by a factor
void multiply(double values[], int size, double factor)
{
    for (int i = 0; i < size; i++)
    {
        values[i] = values[i] * factor;
    }
}
```


Array Parameter Function Example

- Here is the sum function with an array parameter:
 - Notice that to pass one array, it takes two parameters.

```
double sum(double data[], int size)
{
    double total = 0;
    for (int i = 0; i < size; i++)
    {
        total = total + data[i];
    }
    return total;
}
```

Arrays as Parameters and Return Value

If a function can change the size of an array, it should let the caller know the new size by returning it:

```
int read_inputs(double inputs[], int capacity)
{
    //returns the # of elements read, as int
    int current_size = 0;
    double input;
    while (cin >> input)
    {
        if (current_size < capacity)
        {
            inputs[current_size] = input;
            current_size++;
        }
    }
    return current_size;
}
```

Array Parameters in Functions: Calling the Function

- Here is a call to the `read_inputs` function:

```
const int MAXIMUM_NUMBER = 1000;  
double values[MAXIMUM_NUMBER];  
int current_size = read_inputs(values, MAXIMUM_NUMBER);
```

- After the call, the `current_size` variable specifies how many were added.

Function to Fill or Append to an Array

Or it can let the caller know by passing and returning the current size:

```
int append_inputs(double inputs[], int capacity, int current_size)
{
    double input;
    while (cin >> input)
    {
        if (current_size < capacity)
        {
            inputs[current_size] = input;
            current_size++;
        }
    }
    return current_size;
}
```

- *Note this function has the added benefit of either filling an empty array or appending to a partially-filled array*

Constant Array Parameters

- When a function doesn't modify an array parameter, it is considered good style to add the `const` reserved word, like this:

```
double sum(const double values[], int size)
```

- The `const` reserved word helps the reader of the code, making it clear that the function keeps the array elements unchanged.
- If the implementation of the function tries to modify the array, the compiler issues a warning.

The const Parameter Modifier

- Recall: array parameter actually passes address of 1st element
- Function can then modify array!
 - Often desirable, sometimes not!
- Protect array contents from modification
 - Use "const" modifier before array parameter
 - Called "constant array parameter"
 - Tells compiler to "not allow" modifications

Example – function definition

```
// Takes 2 arrays of the same size as input parameters and outputs an array  
// whose elements are the sum of the corresponding elements in the 2 input arrays.
```

```
void addarray(int size,           // IN size of arrays  
              const float A[],   // IN input array  
              const float B[],   // IN input array  
              float C[])         // OUT result array  
{  
    int i;  
    for (i = 0; i < size; i++)  
        C[i] = A[i] + B[i];  
  
} // End of function addarray
```

Example – function call

The function `addarray` could be used as follows:

In `main()`:

```
int one[50], two[50], three[50];
```

```
...
```

```
addarray(50, one, two, three);
```

```
// but also:
```

```
addarray(20, one, two, three);
```

```
// it will only do the addition on the first 20 elements of  
each array
```