C++ Static Arrays, cstrings, and C++ Strings Searching and Sorting Algorithms

In this week

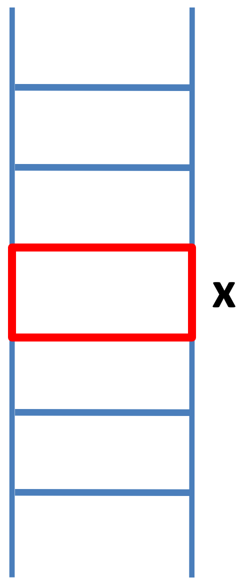
- Declaring, Initializing and Processing Arrays
- Passing Static Arrays to C++ functions
- C++ String data type
- Searching and Sorting Algorithms
- Complexity of Algorithms
- Big-O Notation

Introduction

- So far all the variables we have looked at could store only one value at a time
- For example, in the variable declaration

  double x;

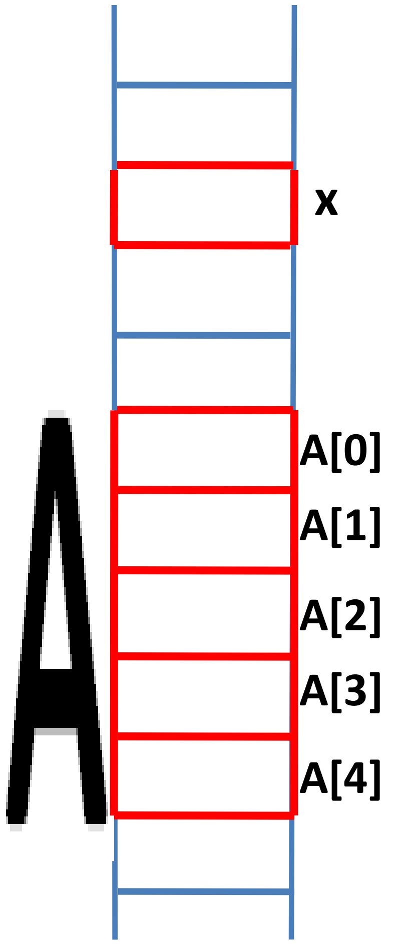
  the variable x can store only one double value at a time
- Of course we can always modify and re-modify the value of the variable **x**, but then the moment we modify its value; its old value will be deleted and only the updated value will be stored in memory
- This makes all the variables we have seen so far to be limited to be able to store only one value at a time
- But what if we want a variable that can store, let's say 100 different double values, at the same time?
- This is where the concept of C++ static arrays data structure comes in to play a role of storing several values all at the same time



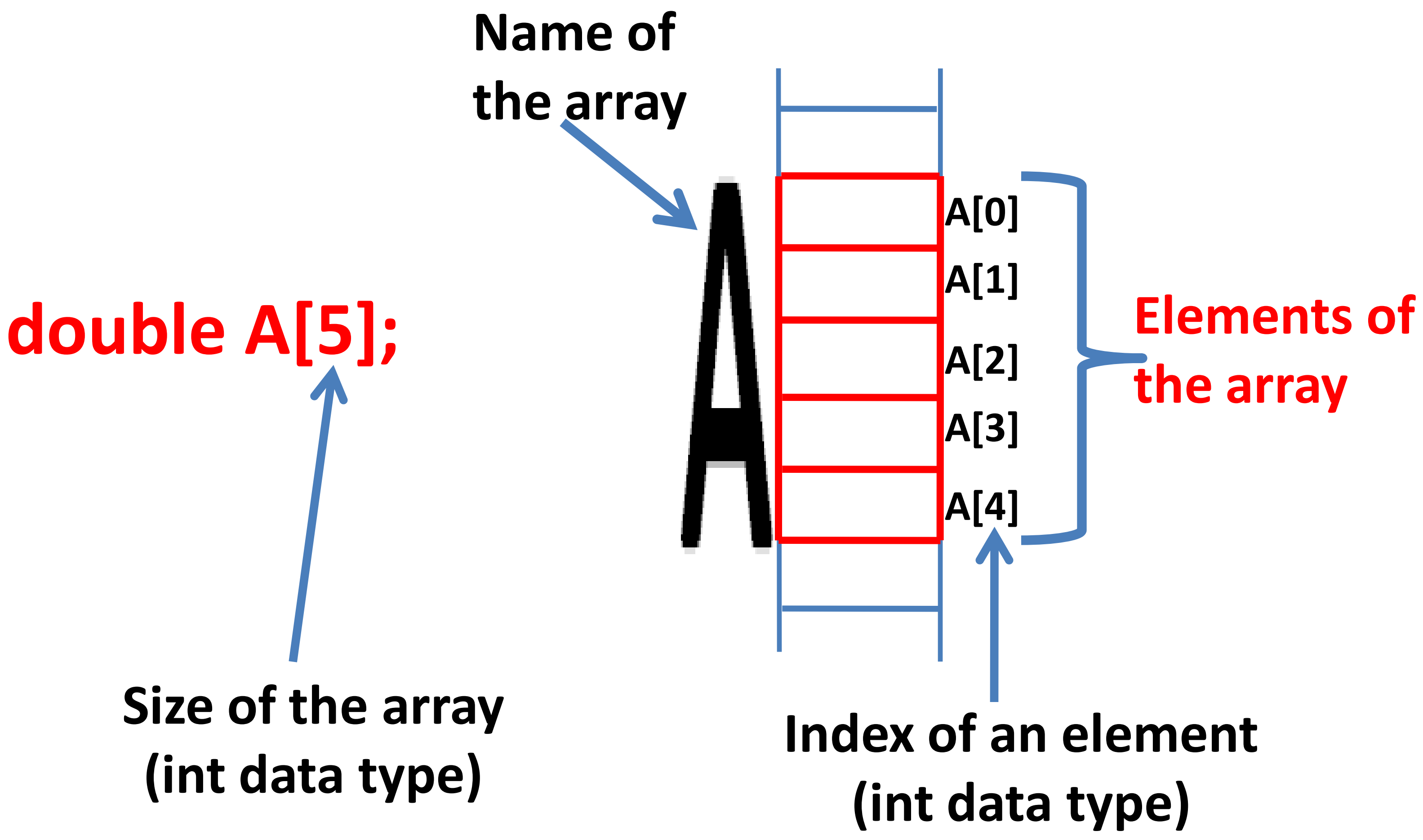
What are C++ Static Arrays?

```
double x;     //double data type variable
double A[5]; //double data type STATIC ARRAY
```

- C++ Static Arrays are consecutive memory locations that share a common name and that store certain predefined number of the same data type values
- The values stored in a C++ static array are called the elements of the array
- The elements of a C++ static array are accessed via the array name and an index placed within square brackets
- The index starts at **0** for the first element and increments by 1 for the subsequent elements



What are C++ Static Arrays?



Static Array Declaration

- As shown above, a **C++ static array** consisting of say **five** elements of type **double** data type is declared as follows

```
double A[5];
```

- In the same way, a **C++ static array** consisting of **ten** elements of type **float** data type is declared as follows:

```
float B[10];
```

- Similarly a **C++ static array** consisting of **one hundred** elements of type **char** data type is declared as:

```
char arr[100];
```

- Also, a **C++ static array** consisting of one **one thousand** elements of **int** data type is declared as:

```
int res[1000];
```

- Finally, a **C++ static array** consisting of **one million** elements of type **boolean** data type is declared as:

```
bool ans[1000000];
```

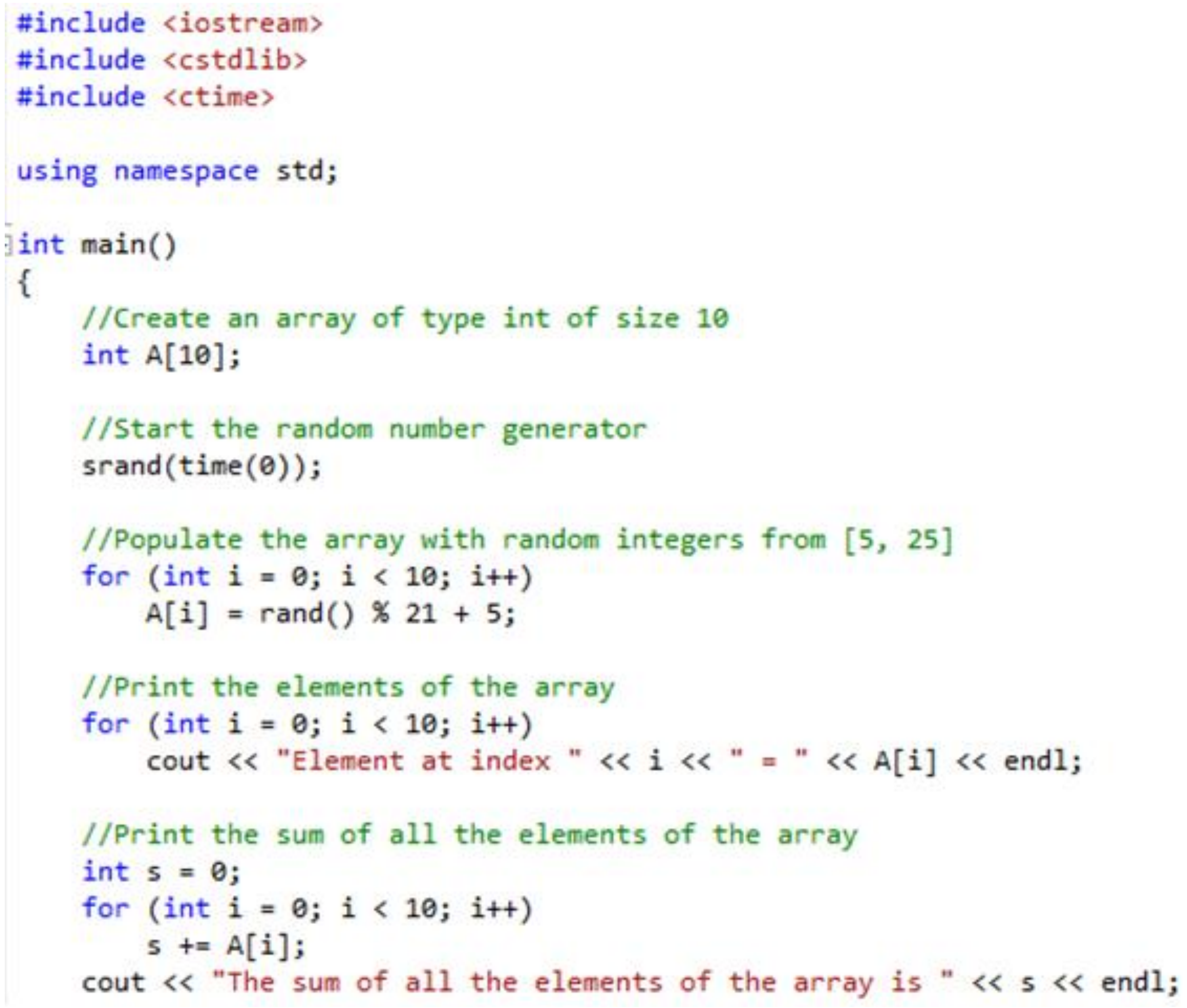
Processing C++ Static Arrays

- Processing C++ static arrays requires accessing elements using their index in order to process them such as assign some values to the elements or perform some computations using the values of the elements
- Index starts at `0` and ends at `size-1` where size is the size of the static array
- Using an index value outside of the range `[0, size-1]` results to **index out of bounds runtime error**
- For example, given a static array A whose elements are already initialized; the statement

  ```
  cout << A[i] << endl;
  ```

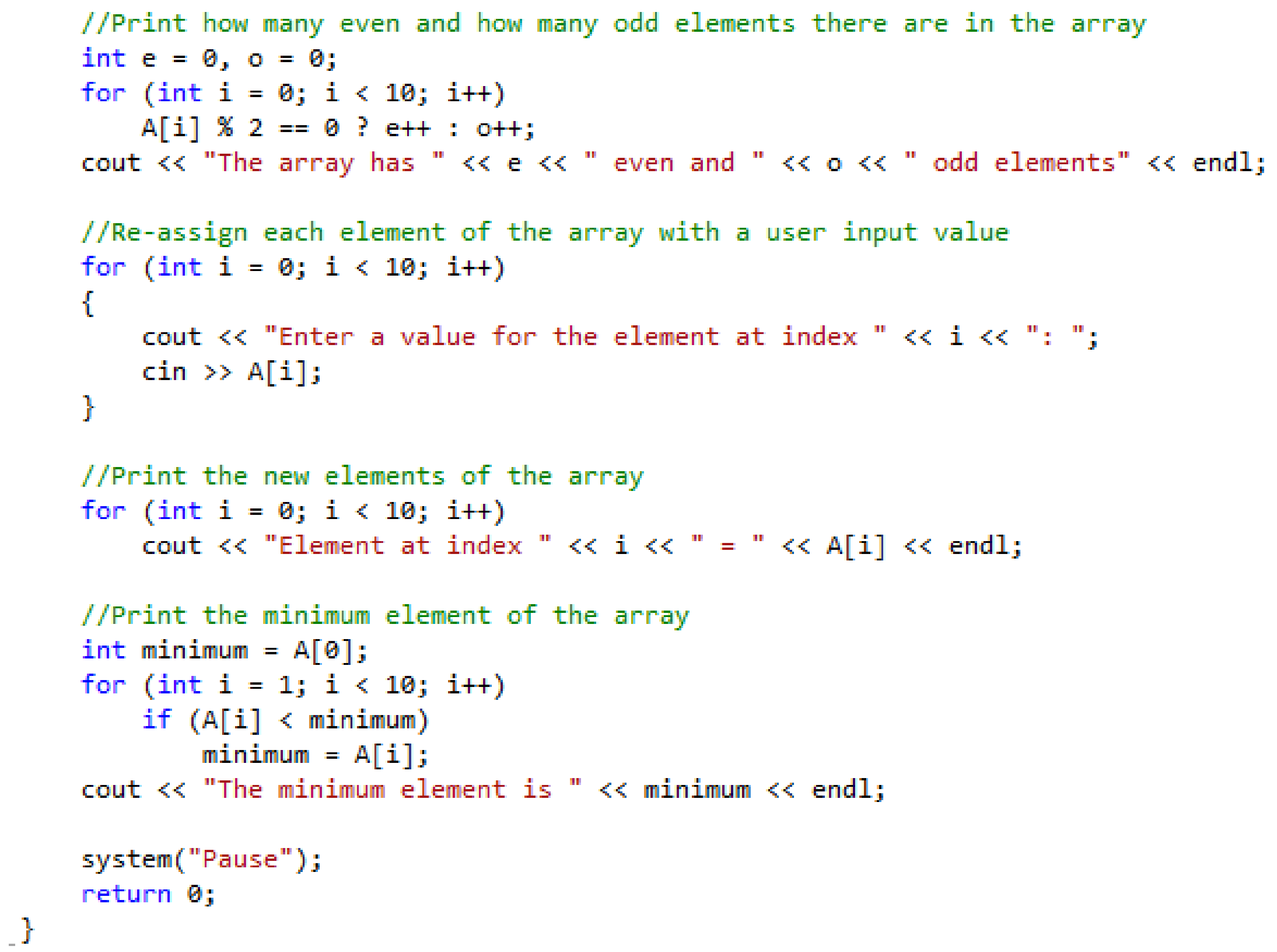
  prints the element at index `i`.
- Unlike in Python, there is no negative indexing in C++

C++ Static Arrays: Example



```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main()
{
    //Create an array of type int of size 10
    int A[10];

    //Start the random number generator
    srand(time(0));

    //Populate the array with random integers from [5, 25]
    for (int i = 0; i < 10; i++)
        A[i] = rand() % 21 + 5;

    //Print the elements of the array
    for (int i = 0; i < 10; i++)
        cout << "Element at index " << i << " = " << A[i] << endl;

    //Print the sum of all the elements of the array
    int s = 0;
    for (int i = 0; i < 10; i++)
        s += A[i];
    cout << "The sum of all the elements of the array is " << s << endl;
```

Processing C++ Static Arrays



```cpp
    //Print how many even and how many odd elements there are in the array
    int e = 0, o = 0;
    for (int i = 0; i < 10; i++)
        A[i] % 2 == 0 ? e++ : o++;
    cout << "The array has " << e << " even and " << o << " odd elements" << endl;

    //Re-assign each element of the array with a user input value
    for (int i = 0; i < 10; i++)
    {
        cout << "Enter a value for the element at index " << i << ": ";
        cin >> A[i];
    }

    //Print the new elements of the array
    for (int i = 0; i < 10; i++)
        cout << "Element at index " << i << " = " << A[i] << endl;

    //Print the minimum element of the array
    int minimum = A[0];
    for (int i = 1; i < 10; i++)
        if (A[i] < minimum)
            minimum = A[i];
    cout << "The minimum element is " << minimum << endl;

    system("Pause");
    return 0;
}
```

C++ Static Arrays Definition

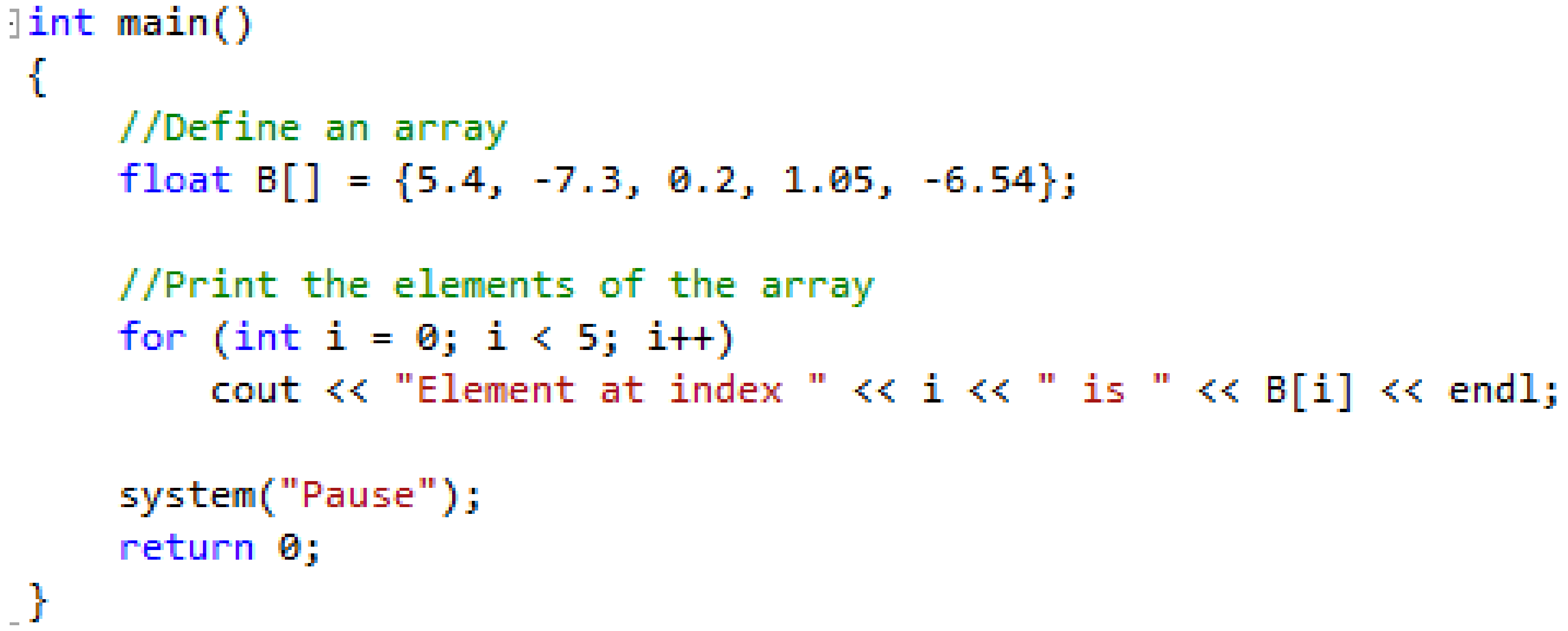
- In the previous example, we first declared an array of size 10 and populated/filled the array either with random integers or from user input data
- If the array data is available during the writing of the code, then we can **define** (declare and initialize) the array as follows:

```
float A[] = {5.4, -7.3, 2.0, 1.05, -6.54};
```

- Now A is a array of float of size 5. The size is determined by C++ based on the number of data.

C++ Static Arrays Definition

- The following C++ code segment demonstrates defining an array of float with 5 float data and then printing the elements of the array



```
int main()
{
    //Define an array
    float B[] = {5.4, -7.3, 0.2, 1.05, -6.54};

    //Print the elements of the array
    for (int i = 0; i < 5; i++)
        cout << "Element at index " << i << " is " << B[i] << endl;

    system("Pause");
    return 0;
}
```

C++ Static Arrays Definition

- **Syntax 1: Data size determines array size**

  `int A[] = {4, 9, 15, 0, 8};`

  This creates an array of five integers and initializes them with the given data. The size 5 is computed by C++
- **Syntax 2: Both data and size specified**

  `int A[5] = {4, 9, 15, 0, 8};`

  This creates an array of five integers and initializes them with the given data
- **Syntax 3: Partial Initialization and Default Initialization**

  `int A[5] = {3, 4, 5};`

  This creates an array of five integers, initializes A[0], A[1], and A[2] with the given data. The remaining elements (A[3] and A[4]) are initialized to zero.
- **Syntax 4: Default initialization**

  `int A[5] = {};`

  This creates an array of five integers and initializes each element to zero.
- **Syntax 5: ERROR!**

  `int A[5] = {1, 2, 3, 4, 5, 6, 7, 8};`

  This is syntax **ERROR** because the number of the data supplied must not exceed the specified size.

Rules for C++ Static Array Declaration

- In C++, the **size** of a static array can be either a **literal value** as shown earlier or an integer type **named constant** as follows:

```
const int SIZE = 30;
float S[SIZE]; ←←←←← CORRECT
```

- Now **S** is a C++ static array with size **30** and can hold up to 30 float data
- Here the **SIZE** variable must be an integer data type **named constant** otherwise it will be a syntax error. See the following examples

```
const int SIZE = rand() % 5 + 10;
int A[SIZE]; ←←←←← SYNTAX ERROR
```

OR

```
int SIZE = 10;
int A[SIZE]; ←←←←← SYNTAX ERROR
```

OR

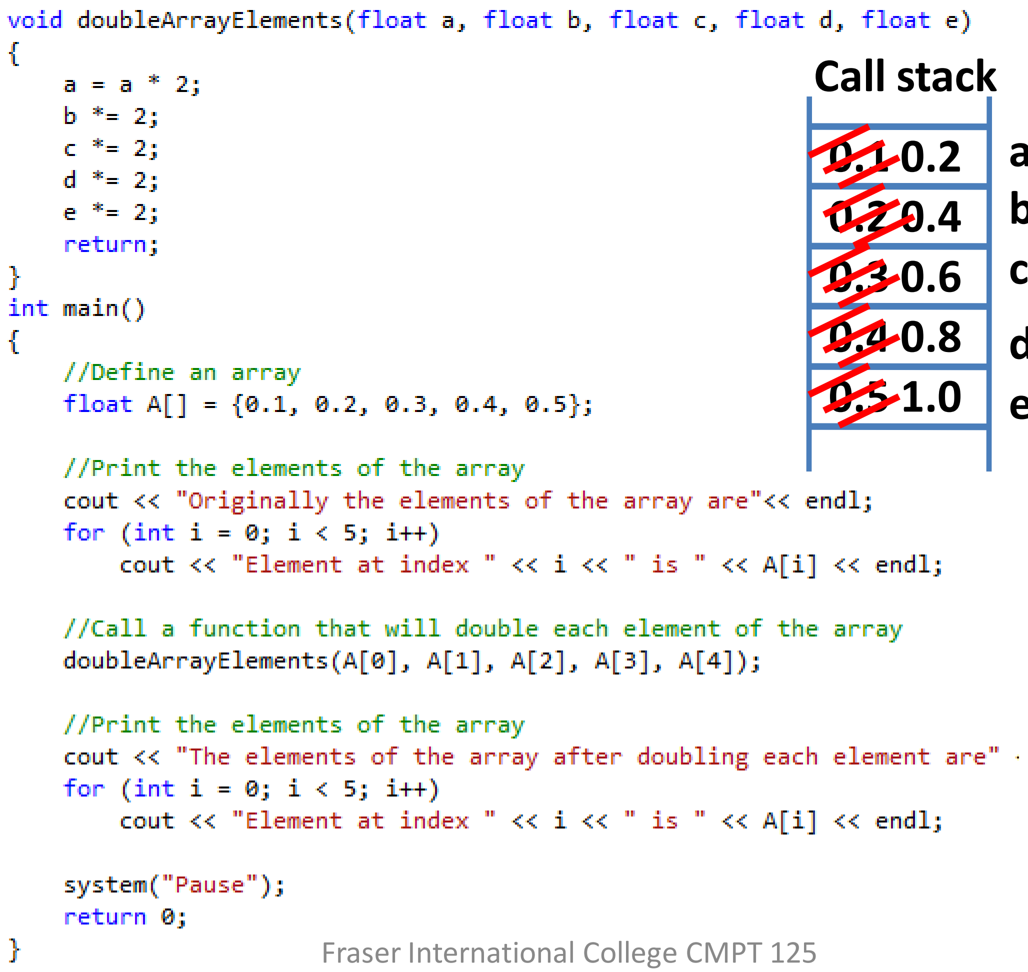
```
int SIZE;
cout << "Enter size: ";
cin >> SIZE;
int A[SIZE]; ←←←←← SYNTAX ERROR
```

Rules for data stored in C++ Static Arrays

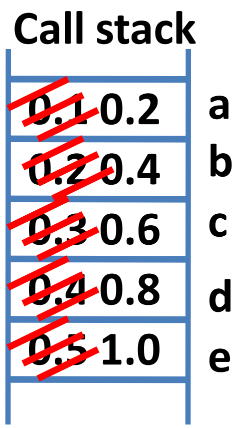
- In C++ the size of static arrays, that is the number of elements the array can hold, is defined at the declaration time
- The array size can not be changed once the array is declared
- This is why the size is required to be either a literal value or a named constant
- The collection of data to be stored in the array must be of the same data type and must match the array declaration data type

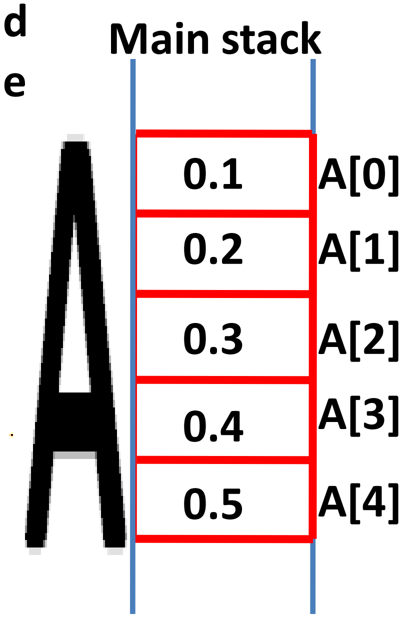
Passing Array Elements to Functions

- The following example demonstrates passing elements of an array to a function by value. The function doubles the elements. **But the Array is not affected, why?**



```
void doubleArrayElements(float a, float b, float c, float d, float e)
{
    a = a * 2;
    b *= 2;
    c *= 2;
    d *= 2;
    e *= 2;
    return;
}
int main()
{
    //Define an array
    float A[] = {0.1, 0.2, 0.3, 0.4, 0.5};

    //Print the elements of the array
    cout << "Originally the elements of the array are"<< endl;
    for (int i = 0; i < 5; i++)
        cout << "Element at index " << i << " is " << A[i] << endl;

    //Call a function that will double each element of the array
    doubleArrayElements(A[0], A[1], A[2], A[3], A[4]);

    //Print the elements of the array
    cout << "The elements of the array after doubling each element are"
    for (int i = 0; i < 5; i++)
        cout << "Element at index " << i << " is " << A[i] << endl;

    system("Pause");
    return 0;
}
```





Passing C++ Static Arrays to Function

- Similarly, we can pass an array variable to a function as an argument
- However, an array argument passes to a function **NOT by value**
- Rather a static array is always passed by a special kind of parameter passing that makes it possible to change its values permanently inside functions
- Therefore any change made to a static array inside a function **ALSO CHANGES THE ARRAY IN THE MAIN PROGRAM!!!**

Passing C++ Static Arrays to Function

- The syntax for static array function argument and static array function parameter is as follows:
- **Function Parameter Declaration Syntax**

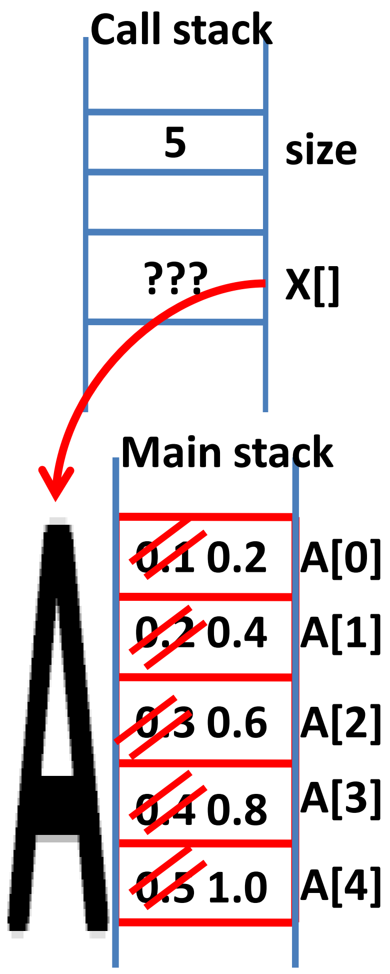
```
void doubleArrayElements(float X[ ], int size)
{
        //Function body
}
```

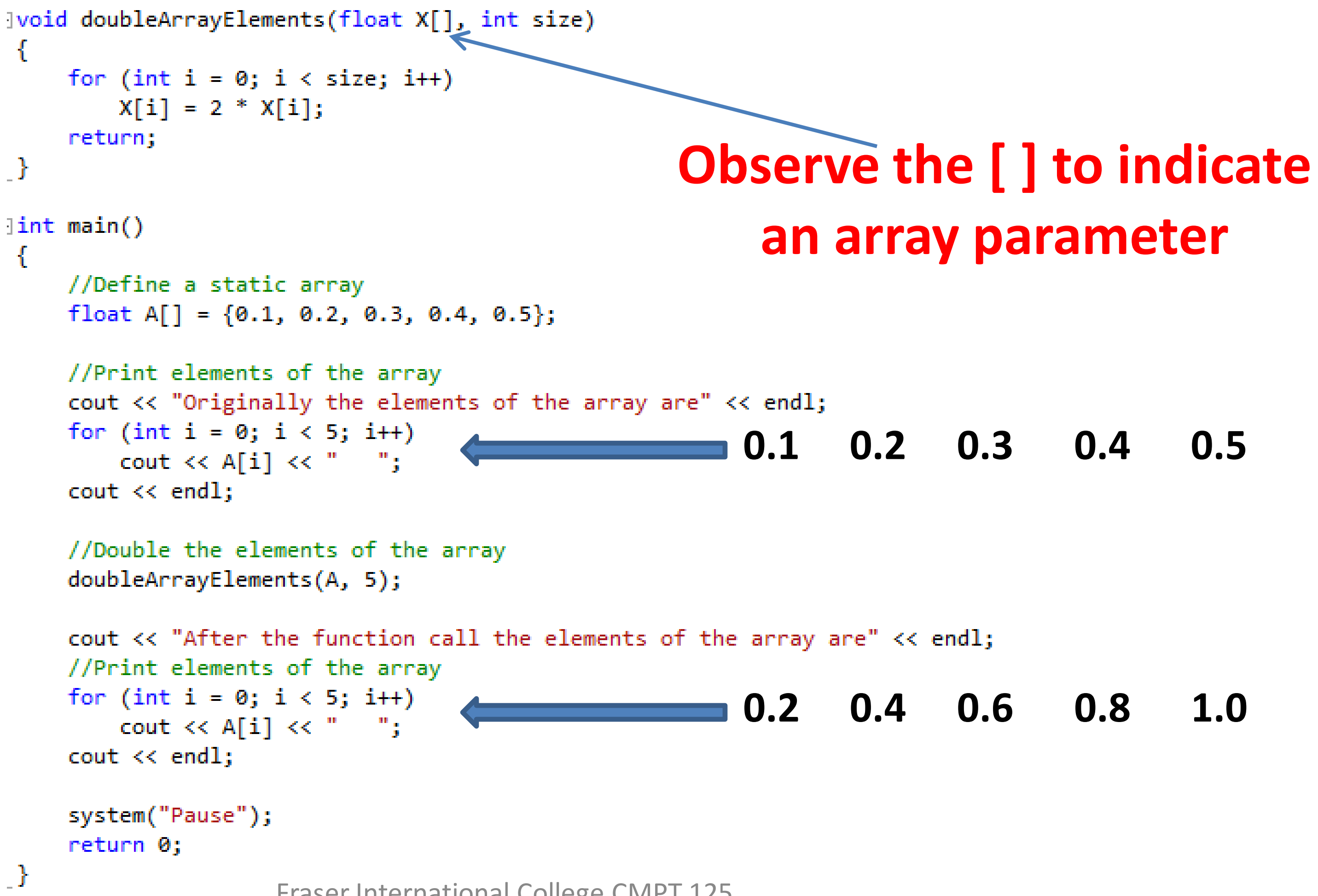
- Function Call Syntax

```
float A[] = {0.1, 0.2, 0.3, 0.4, 0.5};
doubleArrayElements(A, 5);
```

Passing C++ Static Arrays to Function

- The following code demonstrates array passing to functions



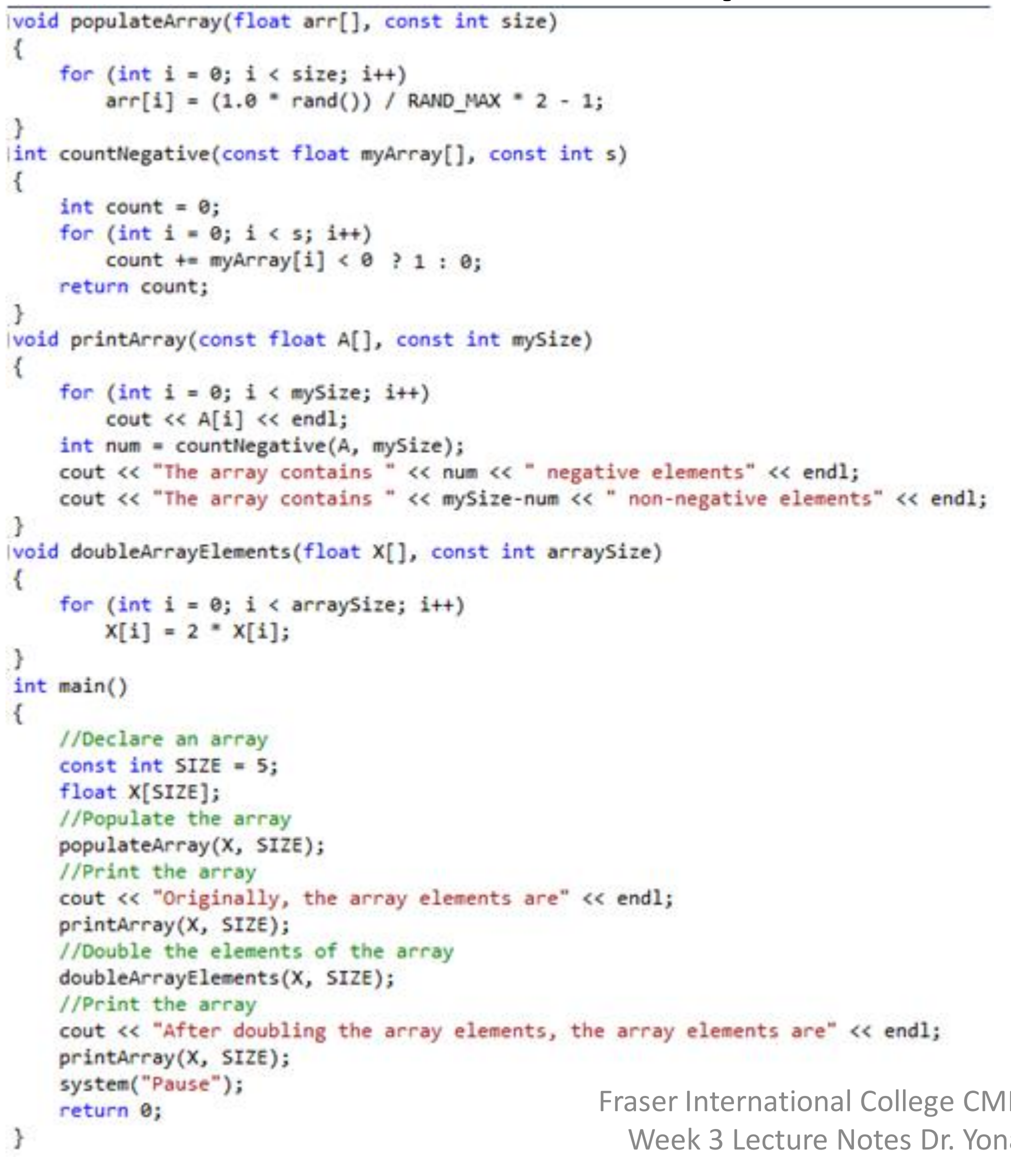


```
void doubleArrayElements(float X[], int size)
{
    for (int i = 0; i < size; i++)
        X[i] = 2 * X[i];
    return;
}

int main()
{
    //Define a static array
    float A[] = {0.1, 0.2, 0.3, 0.4, 0.5};

    //Print elements of the array
    cout << "Originally the elements of the array are" << endl;
    for (int i = 0; i < 5; i++)
        cout << A[i] << "   ";
    cout << endl;

    //Double the elements of the array
    doubleArrayElements(A, 5);

    cout << "After the function call the elements of the array are" << endl;
    //Print elements of the array
    for (int i = 0; i < 5; i++)
        cout << A[i] << "   ";
    cout << endl;

    system("Pause");
    return 0;
}
```

Returning C++ Static Arrays From Function

- Recall that a C++ function can return at most one value
- But a C++ static array contains more than one value
- Therefore we conclude that **a C++ function can not return a static array**
- This is true even if the static array has only one element. We can return the element but we can not return the static array from a function

Passing C++ Static Arrays to Functions Complete Example



```
void populateArray(float arr[], const int size)
{
    for (int i = 0; i < size; i++)
        arr[i] = (1.0 * rand()) / RAND_MAX * 2 - 1;
}
int countNegative(const float myArray[], const int s)
{
    int count = 0;
    for (int i = 0; i < s; i++)
        count += myArray[i] < 0 ? 1 : 0;
    return count;
}
void printArray(const float A[], const int mySize)
{
    for (int i = 0; i < mySize; i++)
        cout << A[i] << endl;
    int num = countNegative(A, mySize);
    cout << "The array contains " << num << " negative elements" << endl;
    cout << "The array contains " << mySize-num << " non-negative elements" << endl;
}
void doubleArrayElements(float X[], const int arraySize)
{
    for (int i = 0; i < arraySize; i++)
        X[i] = 2 * X[i];
}
int main()
{
    //Declare an array
    const int SIZE = 5;
    float X[SIZE];
    //Populate the array
    populateArray(X, SIZE);
    //Print the array
    cout << "Originally, the array elements are" << endl;
    printArray(X, SIZE);
    //Double the elements of the array
    doubleArrayElements(X, SIZE);
    //Print the array
    cout << "After doubling the array elements, the array elements are" << endl;
    printArray(X, SIZE);
    system("Pause");
    return 0;
}
```

A complete C++ main program along with several functions designed to create an array, populate the array, print the array and modify the array elements... as well as demonstrate constant qualifier for variables and parameters.

Rules of Assignment for Static Arrays

- A static array variable can not be re-assigned
- Example:- Given

```
int A[3] = {1,2,3};
int B[3] = {4,5,6};
```

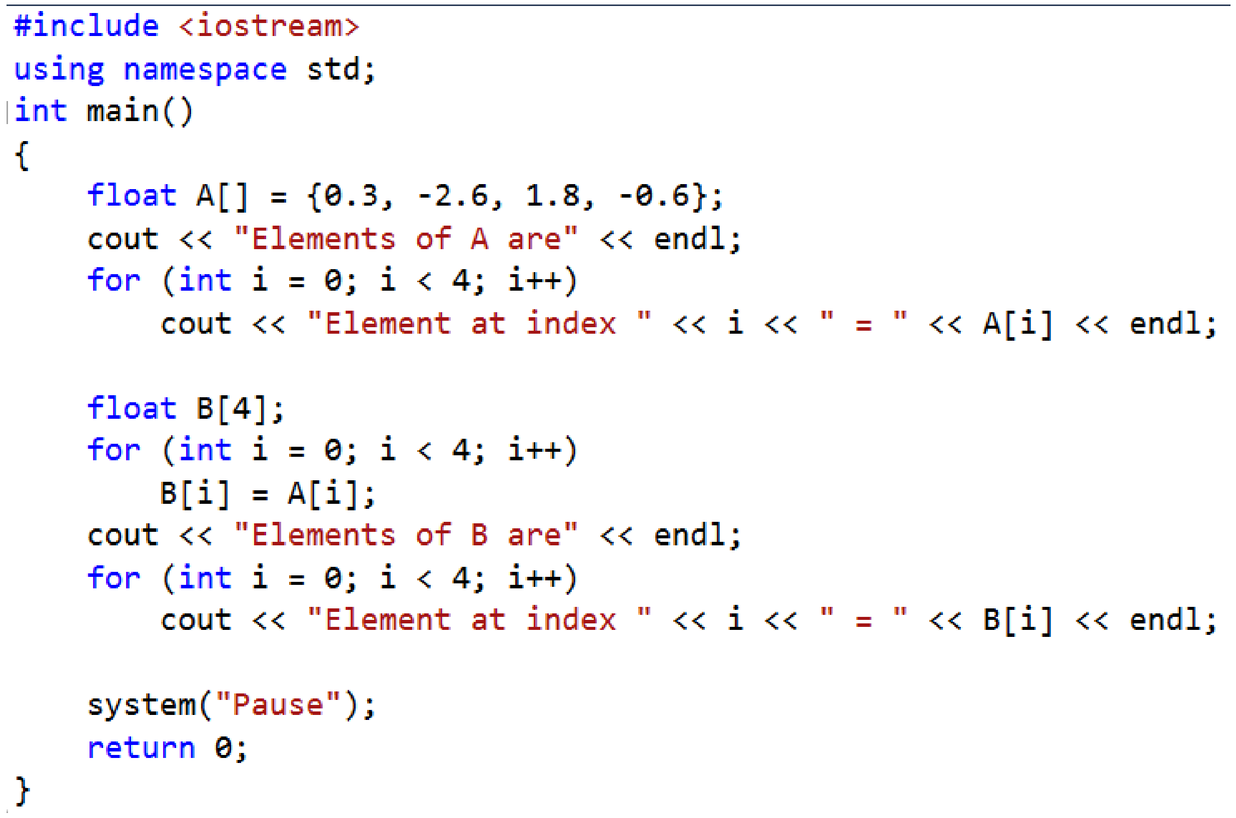
- Then the assignment statement

```
B = A;   //Syntax Error
```

is **syntax error** because **A** can not be re-assigned

Rules of Assignment for Static Array

- Of course we can copy elements of an array to another array as shown in the example below



```
#include <iostream>
using namespace std;
int main()
{
    float A[] = {0.3, -2.6, 1.8, -0.6};
    cout << "Elements of A are" << endl;
    for (int i = 0; i < 4; i++)
        cout << "Element at index " << i << " = " << A[i] << endl;

    float B[4];
    for (int i = 0; i < 4; i++)
        B[i] = A[i];
    cout << "Elements of B are" << endl;
    for (int i = 0; i < 4; i++)
        cout << "Element at index " << i << " = " << B[i] << endl;

    system("Pause");
    return 0;
}
```

char data type arrays

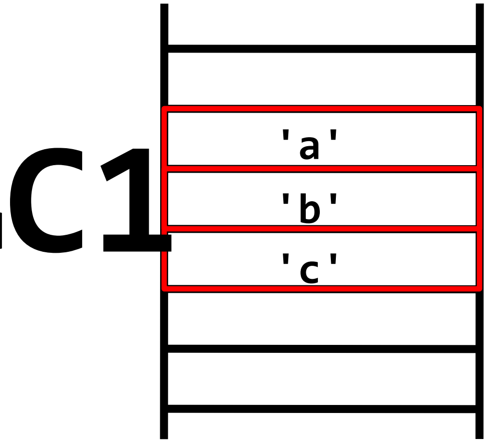
- In C/C++, char data type arrays are treated a little differently than other data types
- For example, consider the char data type array definition

```
char C1[3] = {'a', 'b', 'c'};
```

- We can now process the elements of the static array **C1** in the same way as we process other data types such as int, float, double, bool, etc
- Thus the code fragment

```
    for (int i = 0; i < 3; i++)
        cout << C1[i] << endl;
```

  will print the elements of the array as expected



- Moreover, the code fragment

```
    for (int i = 0; i < 3; i++)
        C1[i]++;
```

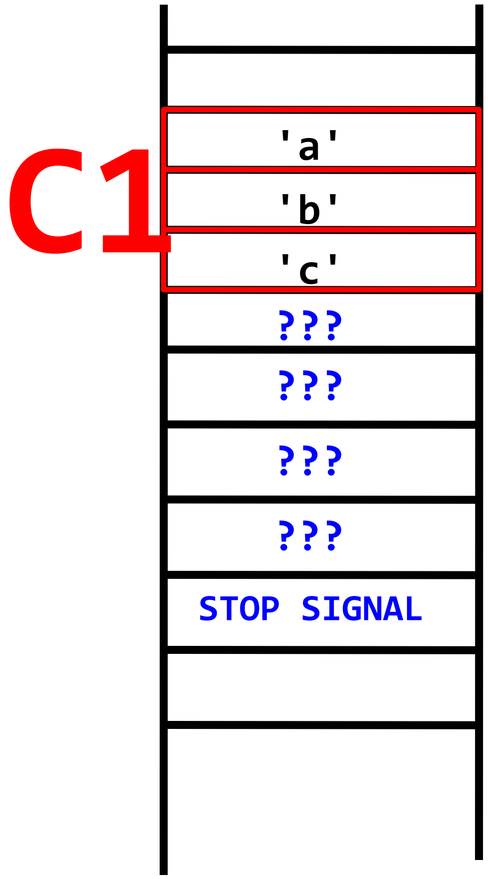
  would modify the elements of the array to **'b'**, **'c'**, and **'d'** respectively.
- So far the char data type array is treated the same as other data types

char data type arrays

- In addition, we could also print all the elements of a ***char data type*** array without indexing as follows

```
char C1[3] = {'a', 'b', 'c'};
cout << C1 << endl;
```

- Unlike with other data types, the cout statement shown above will print all the elements of the array as **abc**
- That is great feature!
- Unfortunately, it will not stop printing after the last element (which is **c**)
- Instead, it will keep on printing values outside of the static array (***which is a runtime error***) until it finds a **STOP** signal
- C/C++ languages are designed to stop printing when they find a special character named **NULL** character (**whose ASCII code is 0 and is written as '\0'**) which is used as a STOP signal
- Thus unless our static array has the NULL character as its last element, then attempting to print all the elements of a char data type static array will always result to a run time error

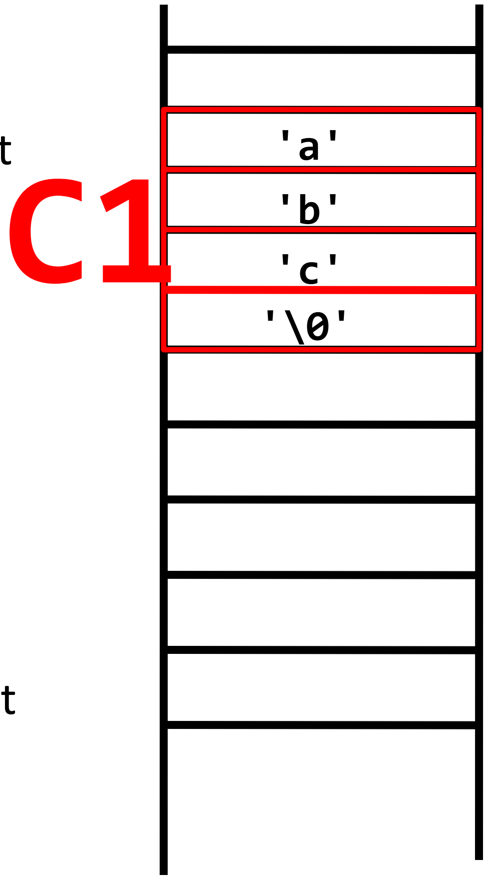


cstrings

- We can therefore create a char data type static array whose last element is a NULL character and then be able to process the elements of the array one by one by indexing as we do with any other data type static arrays or process (such as print) all the elements at once without indexing as follows

```
char C1[4] = {'a', 'b', 'c', '\0'};
cout << C1 << endl;
```

- Now the output will be abc with no runtime error!
- The NULL character is not printed in the cout statement. Instead it is used as a STOP signal to instruct the cout statement to stop printing!
- A char data type array whose last element is a NULL character is known as a **cstring**
- In C programming language, cstrings are used to represent the string data type (although strictly speaking cstring is not a data type; instead it is simply an array of char data type)



Length of cstrings

- The characters in a cstring excluding the NULL character are called the printable characters
- The NULL terminating character is known as the non-printable character
- Thus a cstring is a char data type array containing zero or more printable characters followed by the non-printable NULL character
- The **length** of a cstring is defined as the number of printable characters in the cstring
- Thus the length of the cstring

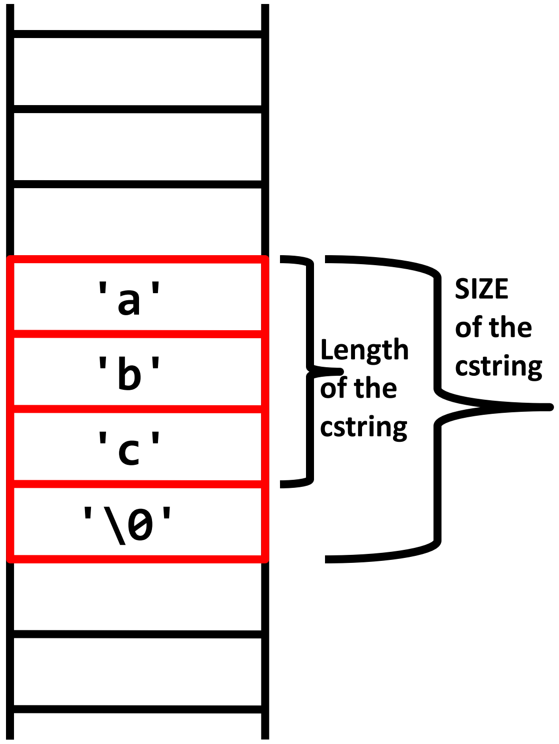
```
    char C1[4] = {'a', 'b', 'c', '\0'};
```

  is 3 (not 4)
- Of course the size of the array is 4 but its length 3

Length of cstrings

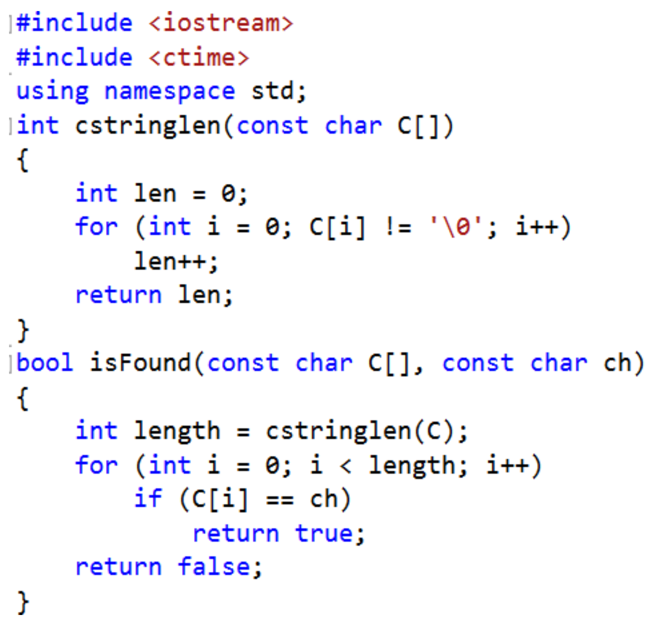
- **Practice Exercise:** Define a C++ function named ***cstringlen*** that takes a cstring argument and returns its length.
- We can compute the length of a cstring easily by counting the elements of the cstring until the NULL character is found as shown below

```
#include <iostream>
using namespace std;
int cstringlen(const char C[])
{
    int len = 0;
    for (int i = 0; C[i] != '\0'; i++)
        len++;
    return len;
}
int main()
{
    char C1[] = {'a', 'b', 'c', '\0'};
    cout << "The cstring is " << C1 << endl;
    int length = cstringlen(C1);
    cout << "The length of the cstring is " << length << endl;
    for (int i = 0; i < length; i++)
        C1[i]++;
    cout << "After modification, the cstring is " << C1 << endl;
    system("Pause");
    return 0;
}
```

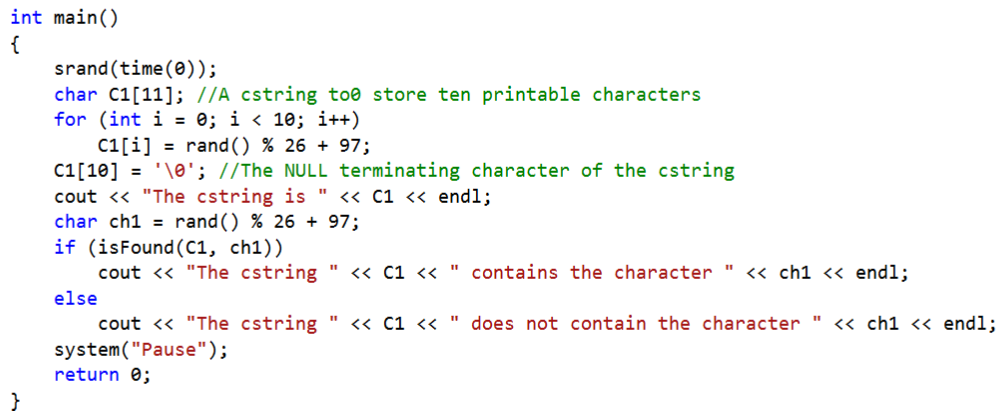


Processing cstrings

- cstrings are processed by first computing their length, for example using the cstringlen function definition given above, and then use the length to process the printable characters in the cstring in a loop. See below



```cpp
#include <iostream>
#include <ctime>
using namespace std;
int cstringlen(const char C[])
{
    int len = 0;
    for (int i = 0; C[i] != '\0'; i++)
        len++;
    return len;
}
bool isFound(const char C[], const char ch)
{
    int length = cstringlen(C);
    for (int i = 0; i < length; i++)
        if (C[i] == ch)
            return true;
    return false;
}
```



```cpp
int main()
{
    srand(time(0));
    char C1[11]; //A cstring to0 store ten printable characters
    for (int i = 0; i < 10; i++)
        C1[i] = rand() % 26 + 97;
    C1[10] = '\0'; //The NULL terminating character of the cstring
    cout << "The cstring is " << C1 << endl;
    char ch1 = rand() % 26 + 97;
    if (isFound(C1, ch1))
        cout << "The cstring " << C1 << " contains the character " << ch1 << endl;
    else
        cout << "The cstring " << C1 << " does not contain the character " << ch1 << endl;
    system("Pause");
    return 0;
}
```

Processing cstrings

- The cstrings can be defined using the static arrays definition syntax as shown earlier as in

```
char C1[] = {'a', 'b', 'c', '\0'};
```

- In addition, the same cstring can be defined as

```
char C2[] = "abc";
```

- Now C2 is a cstring whose elements are `'a'`, `'b'`, `'c'`, and `'\0'`
- In this case, the compiler will convert the characters inside the double quotes to characters enclosed in curly braces and adds the NULL terminating character
- **Thus the cstrings C1 and C2 are identical**

cstring built-in functions

- C programming language has some useful libtrary functions to process cstrings such as
- ➢ **`strlen(C)`**
  - ❖ This function returns the length of the cstring argument C
- ➢ **`strcmp(C1, C2)`**
  - ❖ This function compares the two cstrings C1 and C2 and returns 0 if they are equal, returns 1 is C1 is greater than C2, and returns -1 otherwise
- ➢ **`strcpy(C1, C2)`**
  - ❖ This function copies all the characters of C2 including the NULL terminating character to C1. The size of the C1 array (or cstring) must be such that it is big enough to store all the characters of C2 including the NULL terminating character; otherwise a run time will occur.

C++ string Data Type

- **Definition:-** A C++ string is a data type that can hold **ZERO** or **MORE** **characters** enclosed in between double quotes
- In order to work with C++ strings, we need the include directive

  #include <string>

- Now we can declare, initialize and define string variables as follows:

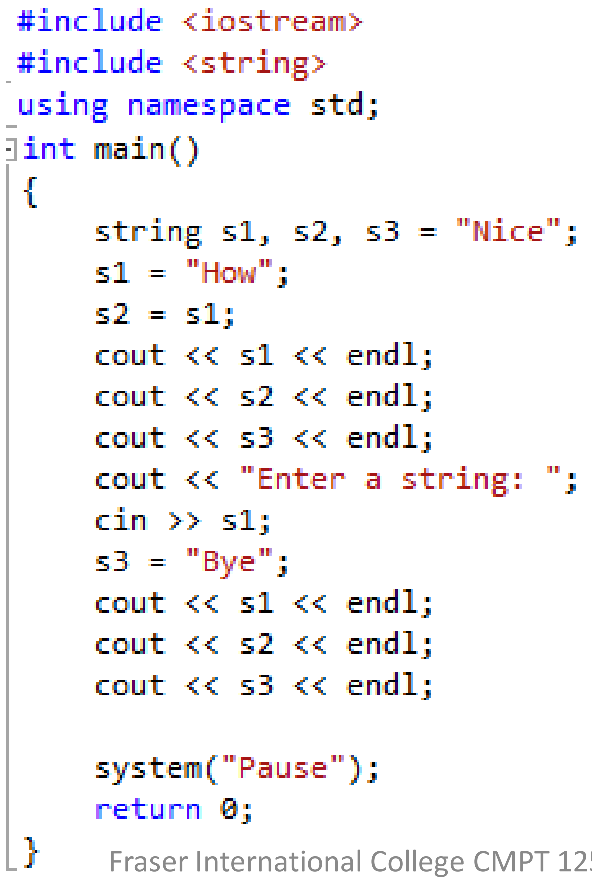
```
string s;                  //string declaration
s = "This is nice";        //string initialization
string  x = "CMPT $130#";  //string definition
```

Working C++ strings

- Just like any simple data type such as int, float, double, char, bool, etc; we can
  - Print strings with **cout** command,
  - Modify the values of string variables with new string values,
  - Read string values from the keyboard using **cin** command,
  - Compare strings,
  - Pass string variables to functions
  - Return strings from functions

Working C++ strings

- The following program demonstrates declaration, initialization, definition, modification, printing and reading in strings



```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string s1, s2, s3 = "Nice";
    s1 = "How";
    s2 = s1;
    cout << s1 << endl;
    cout << s2 << endl;
    cout << s3 << endl;
    cout << "Enter a string: ";
    cin >> s1;
    s3 = "Bye";
    cout << s1 << endl;
    cout << s2 << endl;
    cout << s3 << endl;

    system("Pause");
    return 0;
}
```

The output will be

```
How
How
Nice
Enter a string: CMPT↵
CMPT
How
Bye
```

Empty C++ strings

- A string that contains no characters (i.e. zero characters) is called empty string

```
string s1 = "";
string s2;
```

- Now, we see that s1 is empty string
- Also s2 is empty string. That is when we declare a string variable, C++ automatically initializes it to an empty string

Length of C++ strings

- The number of characters in a string variable is known as the length of the string
- Moreover the string data type has special function (also known as a **method**) that tell us the length of a string variable

```
string s1 = "Yonas", s2;
int a = s1.length();
cout << "The length of " << s1 << " is " << a << endl;
cout << "The length of " << s2 << " is " << s2.length() << endl;
```

- Now the value of **a** is the length of s1 which is **5**
- **Empty string has length zero! Therefore the second line of output will print 0 for the length of s2**

C++ strings concatenation

- C++ strings can be concatenated (added) together to create a new string that has all the characters of the concatenated strings
- We can also **concatenate** a character to a string
- C++ strings concatenation is performed with a + operator
- Example:-

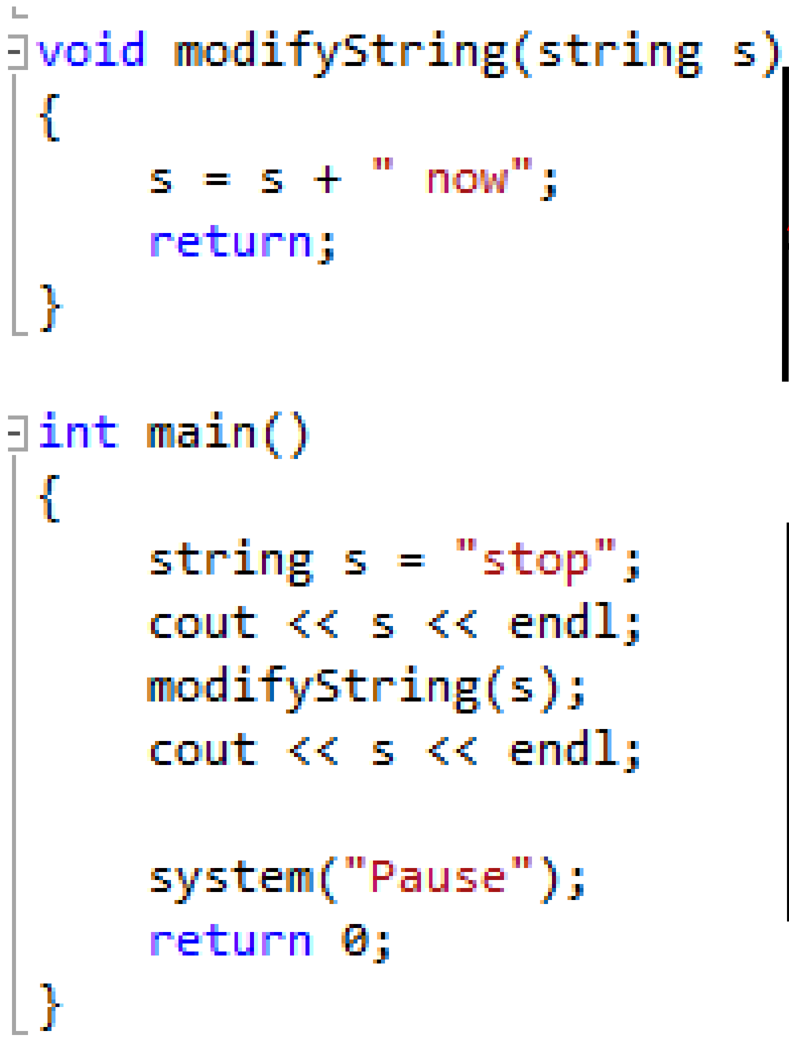
```
string s1 = "stop";
string s2 = "now";
char c = ' ';
string s3 = s1 + c + s2 + " please";
cout << s3 << endl;
```

- The output will be **stop now please**

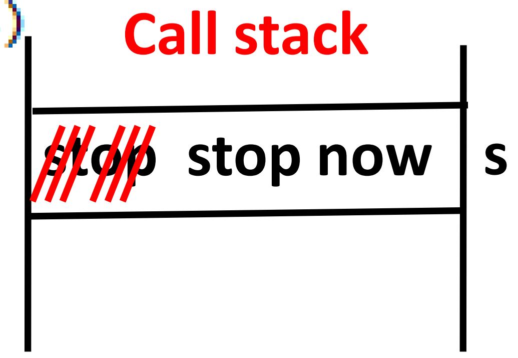
Passing C++ strings to functions

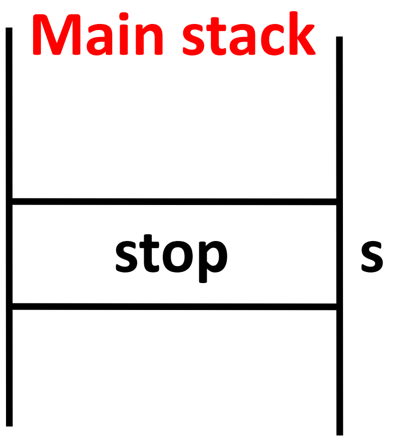
- Parameter passing of C++ strings follows the rule of parameter passing of simple data types such as int, doule, bool,…
- Therefore C++ strings pass to functions by value
- Hence any modification made to a string parameter in a function does not affect the string in the calling function such as main
- Analyze the following program and determine its output

Passing C++ strings to functions



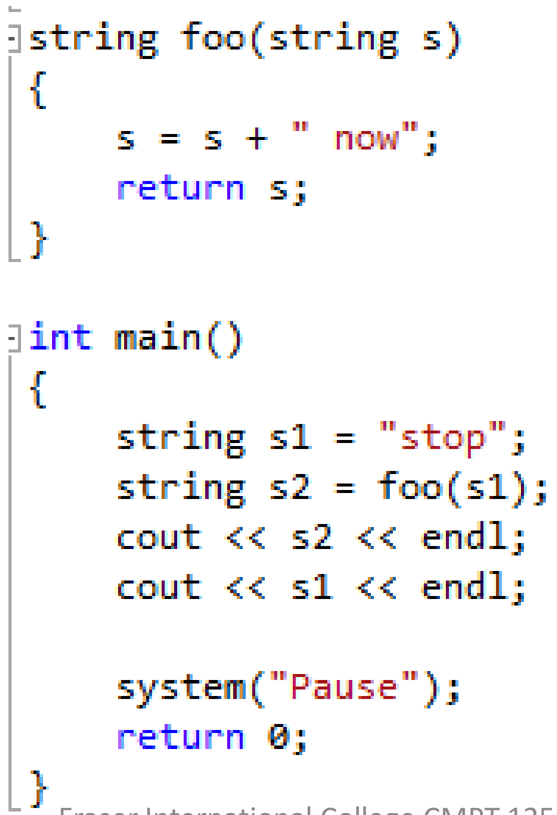
```
void modifyString(string s)
{
    s = s + " now";
    return;
}

int main()
{
    string s = "stop";
    cout << s << endl;
    modifyString(s);
    cout << s << endl;

    system("Pause");
    return 0;
}
```





Returning C++ strings from functions

- We can also return C++ strings from functions
- Analyze the following program and determine its output



```
string foo(string s)
{
    s = s + " now";
    return s;
}

int main()
{
    string s1 = "stop";
    string s2 = foo(s1);
    cout << s2 << endl;
    cout << s1 << endl;

    system("Pause");
    return 0;
}
```

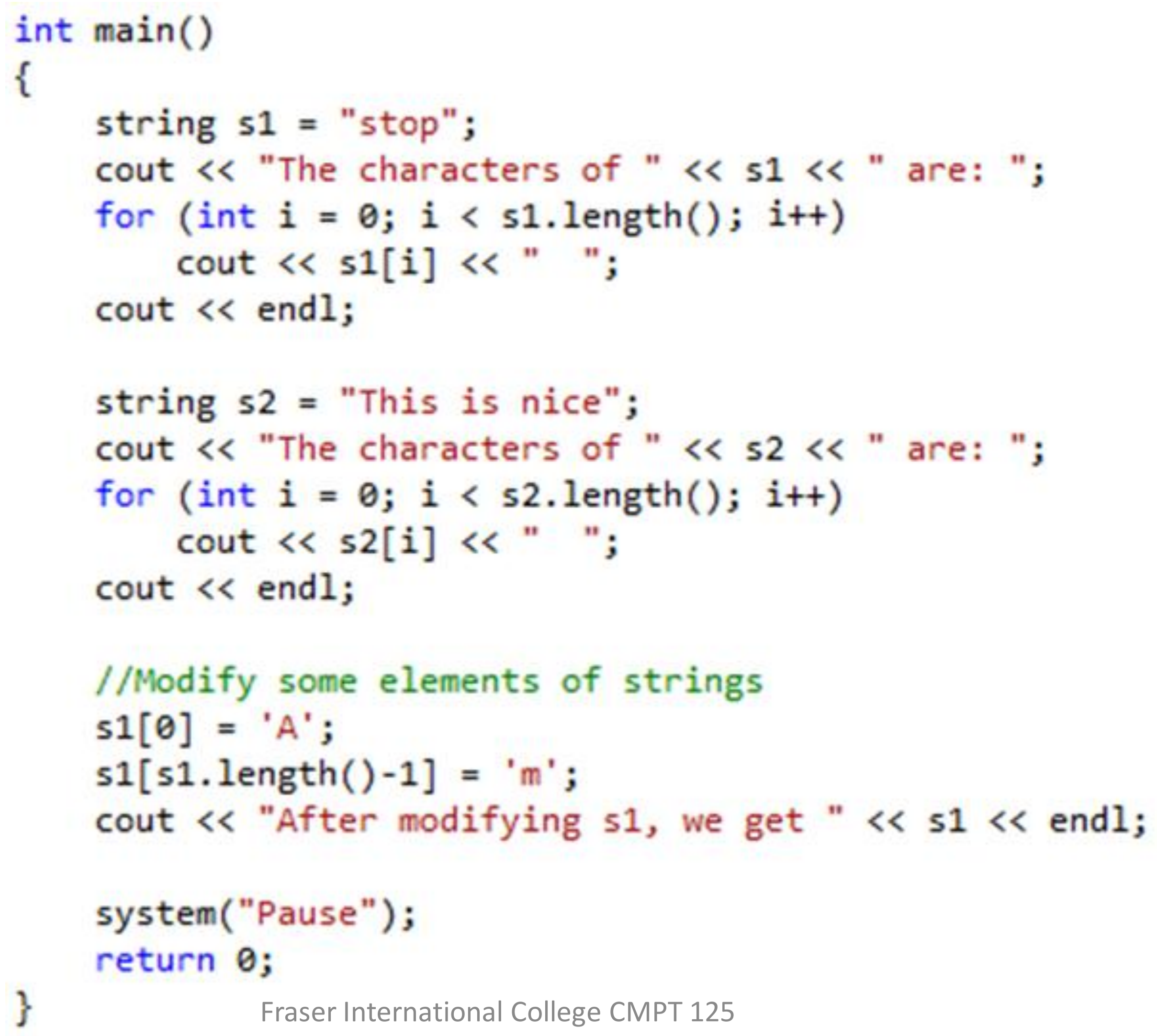
C++ strings processing

- Each character of a string variable can be accessed by the string name and an index placed inside a square bracket
- **Index starts from 0 and goes up to (length-1)**
- Each element of a string is a **char** data type!
- With such indexing, we can now access each element of the string independently so that we can print an element, modify an element by assigning it a new character, read a character from keyboard and store it in an element, modify an element, pass an element to a function, and etc

C++ strings processing

You see that you can print the characters of a string one by one using a loop or print all of them together by printing the string variable.

Can you do the same with arrays?

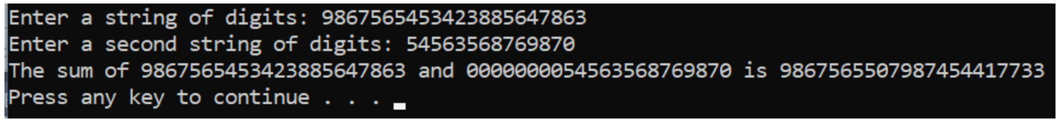


```cpp
int main()
{
    string s1 = "stop";
    cout << "The characters of " << s1 << " are: ";
    for (int i = 0; i < s1.length(); i++)
        cout << s1[i] << "  ";
    cout << endl;

    string s2 = "This is nice";
    cout << "The characters of " << s2 << " are: ";
    for (int i = 0; i < s2.length(); i++)
        cout << s2[i] << "  ";
    cout << endl;

    //Modify some elements of strings
    s1[0] = 'A';
    s1[s1.length()-1] = 'm';
    cout << "After modifying s1, we get " << s1 << endl;

    system("Pause");
    return 0;
}
```

C++ strings concatenation

- The program below demonstrates addition of numbers stored as C++ strings

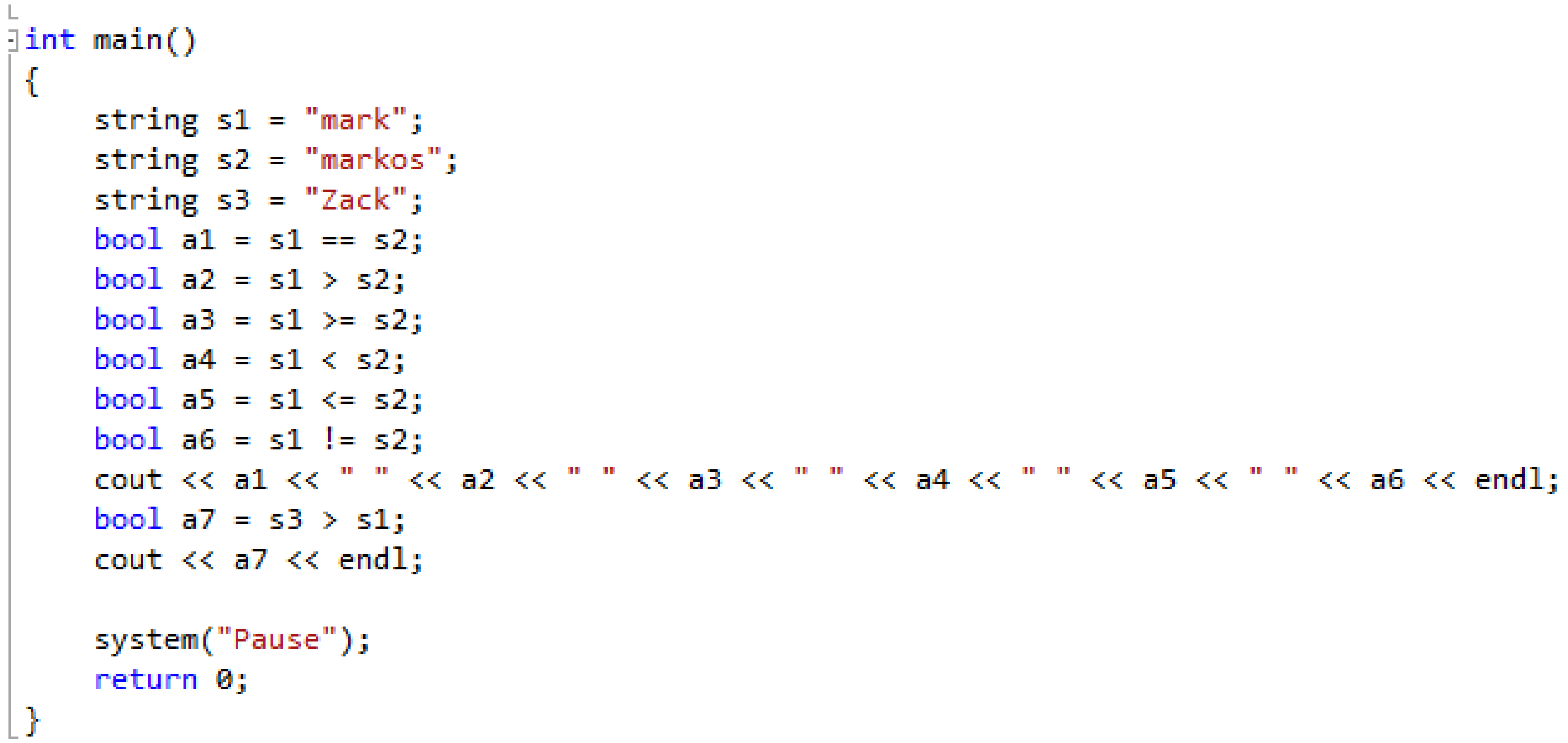
```cpp
#include <iostream>
#include <string>
using namespace std;
int main()
{
   //This program adds two decimal numbers stored as C++ strings
   string s1, s2;
   cout << "Enter a string of digits: ";
   cin >> s1;
   cout << "Enter a second string of digits: ";
   cin >> s2;
   //Step 1. Make their lengths equal
   while (s1.length() < s2.length())
      s1 = '0' + s1;
   while (s2.length() < s1.length())
      s2 = '0' + s2;
   //Step 2. Add s1 and s2 knowing that they now have equal lengths
   string s3 = "";
   int carry = 0;
   for (int i = s1.length()-1; i >= 0; i--)
   {
      int sum = (s1[i] - '0') + (s2[i] - '0') + carry;
      char char_digit = (sum % 10) + '0';
      s3 = char_digit + s3;
      carry = sum / 10;
   }
   if (carry > 0)
   {
      char char_digit = carry + '0';
      s3 = char_digit + s3;
   }
   cout << "The sum of " << s1 << " and " << s2 << " is " << s3 << endl;

   system("Pause");
   return 0;
}
```



```
Enter a string of digits: 9867565453423885647863
Enter a second string of digits: 54563568769870
The sum of 9867565453423885647863 and 0000000054563568769870 is 9867565507987454417733
Press any key to continue . . .
```

C++ strings comparison

- C++ knows how to compare strings! Comparison is done character by character on corresponding indexes. The character comparison uses the ascii code for comparison…



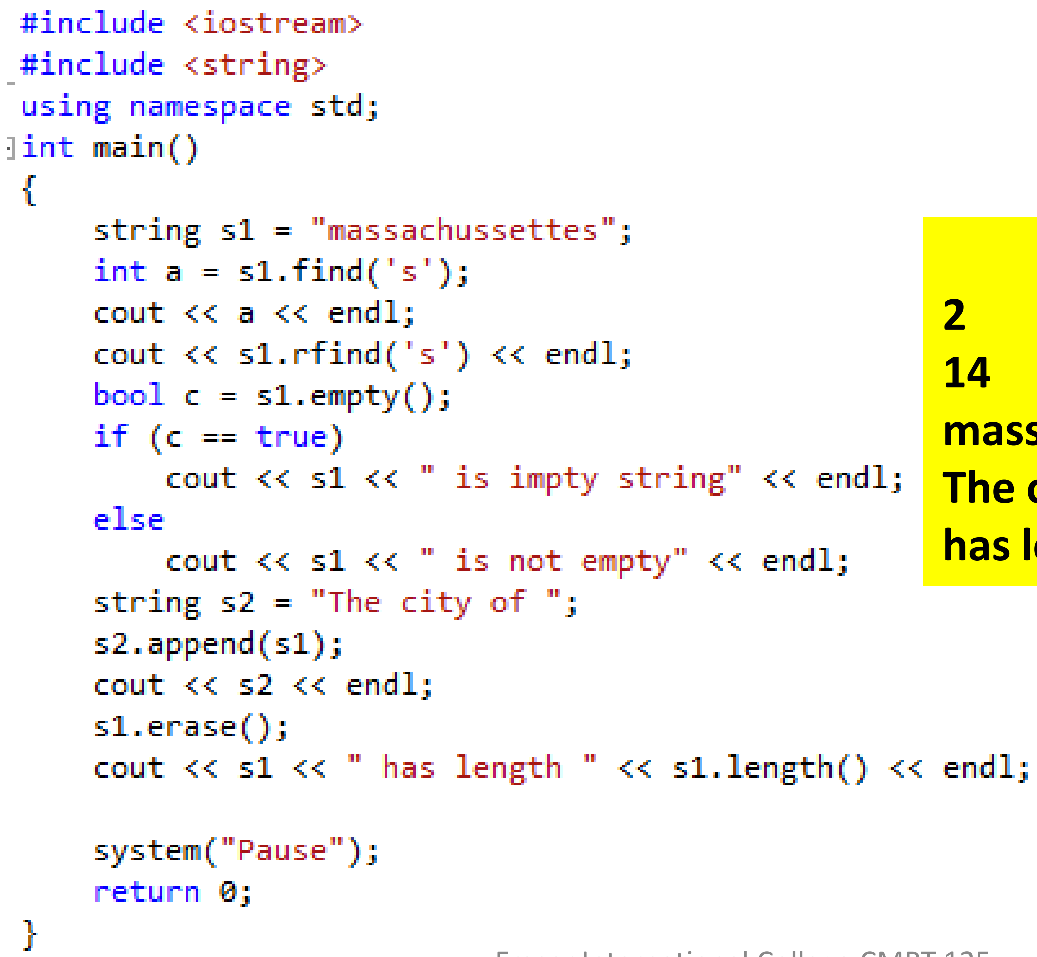
```cpp
int main()
{
    string s1 = "mark";
    string s2 = "markos";
    string s3 = "Zack";
    bool a1 = s1 == s2;
    bool a2 = s1 > s2;
    bool a3 = s1 >= s2;
    bool a4 = s1 < s2;
    bool a5 = s1 <= s2;
    bool a6 = s1 != s2;
    cout << a1 << " " << a2 << " " << a3 << " " << a4 << " " << a5 << " " << a6 << endl;
    bool a7 = s3 > s1;
    cout << a7 << endl;

    system("Pause");
    return 0;
}
```

C++ string methods

- C++ string has many more methods that give some information about a string variable
- Few of these methods are:
  - **find**:- returns the first index of a character argument in a string. Returns **-1** if the character is not found in the string
  - **rfind**:- returns the last index of a character argument in a string. Returns **-1** if the character is not found in the string
  - **empty**:- returns true if a string is empty; otherwise returns false
  - **append**:- appends a string or a character argument to a string
  - **erase**:- deletes all the characters of a string
- More methods at **www.cplusplus.com/reference/string/string/**

C++ string methods

- Consider the following program and determine its output



```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string s1 = "massachussettes";
    int a = s1.find('s');
    cout << a << endl;
    cout << s1.rfind('s') << endl;
    bool c = s1.empty();
    if (c == true)
        cout << s1 << " is impty string" << endl;
    else
        cout << s1 << " is not empty" << endl;
    string s2 = "The city of ";
    s2.append(s1);
    cout << s2 << endl;
    s1.erase();
    cout << s1 << " has length " << s1.length() << endl;

    system("Pause");
    return 0;
}
```

The output will be

2
14
massachussettes is not empty
The city of massachussettes
has length 0

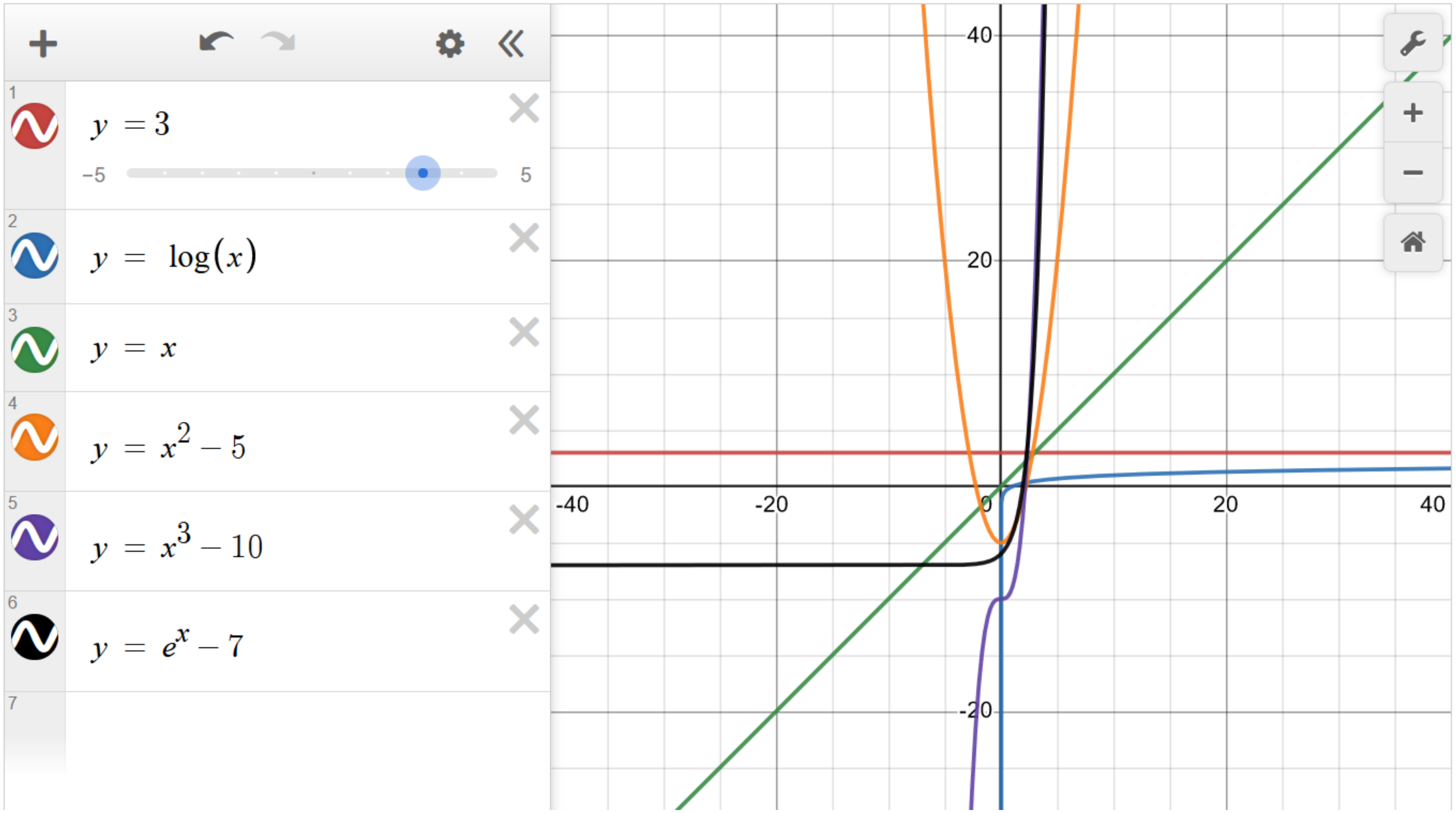
Searching and Sorting Algorithms

- We now look at different searching and sorting algorithms that are commonly used in Computer Science
- We also look at the complexity of these algorithms
- The complexity of algorithms is used to classify algorithms according to how their run time or space requirements grow as the input size grows
- We start by describing some common mathematical functions and their asymptotic growth as the input size grows to infinity

Class of Mathematical Functions

- Mathematical functions are grouped according to how fast they grow into several classes
- Given some appropriate real numbers a, b, c, and d; some of the common class of functions are
  - Constant functions: **$f(x) = a$**
  - Logarithmic functions: **$f(x) = a \log_b x$ ….. (assume $b > 1$)**
  - Linear functions: **$f(x) = ax + b$**
  - Quadratic functions: **$f(x) = ax^2 + bx + c$**
  - Cubic functions: **$f(x) = ax^3 + bx^2 + cx + d$**
  - Exponential functions: **$f(x) = b\, a^x$ ….. (assume $a > 1$)**

Growth of Mathematical Functions



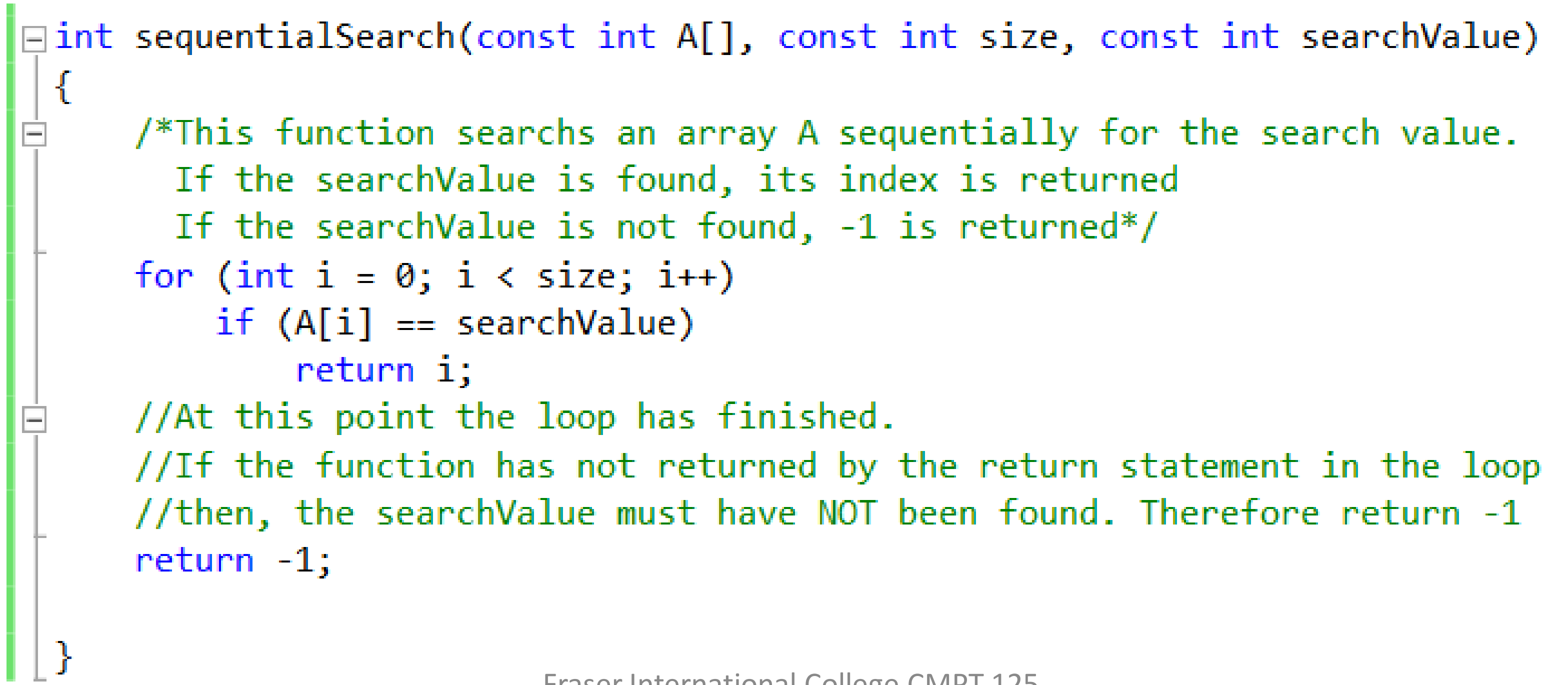
1. $y = 3$
2. $y = \log(x)$
3. $y = x$
4. $y = x^2 - 5$
5. $y = x^3 - 10$
6. $y = e^x - 7$

Big-O Notation

- Big-O notation, is a notation used to denote the asymptotic growth of functions in a compact manner as follows
  - Constant functions: **$f(x) = O(1)$**
  - Logarithmic functions: **$f(x) = O(\log x)$**
  - Linear functions: **$f(x) = O(x)$**
  - Quadratic functions: **$f(x) = O(x^2)$**
  - Cubic functions: $f(x) =$ **$O(x^3)$**
  - Exponential functions: **$f(x) = O(e^x)$**

Sequential Search Algorithm

- Search for a value in an array sequentially!
  - ➢ The following code demonstrates searching a C++ static array sequentially for a specific value



```cpp
int sequentialSearch(const int A[], const int size, const int searchValue)
{
    /*This function searchs an array A sequentially for the search value.
      If the searchValue is found, its index is returned
      If the searchValue is not found, -1 is returned*/
    for (int i = 0; i < size; i++)
        if (A[i] == searchValue)
            return i;
    //At this point the loop has finished.
    //If the function has not returned by the return statement in the loop
    //then, the searchValue must have NOT been found. Therefore return -1
    return -1;

}
```

Sequential Search

```
int main()
{
    const unsigned int size = 300000000;
    int A[size];
    srand(time(0));

    //Fill the array
    for (int i = 0; i < size; i++)
        A[i] = rand() % 200;


    //Perform sequencial Search
    int value = 200;    //Select a number not found in the array
    int index = sequencialSearch(A, size, value);

    //Print concluding remarks
    if (index == -1)
        cout << value << " not found in the array." << endl;
    else
        cout << value << " found in the array at index " << index << endl;

    system("Pause");
    return 0;
}
```

Sequential Search Running Time

- Algorithm description
  - We compare the elements of the array with the search value; one by one starting from the first element and going up to the last
  - If we find an element equal to the search value, then we stop searching and return the index
  - If none of the elements of the array is equal to the search value; then we return -1, to mean the search value was not found in the array

Sequential Search Running Time

- Given an algorithm, it is customary to identify a **critical operation** of the algorithm in order to count how many times that critical operation will be executed for a given input array size
- The critical operation is defined as the operation that determines as to when the algorithm will stop execution
- The count of the critical operation will therefore provide a guideline as the **time** needed for the algorithm to perform the required computation
- In the case of the Sequential Search algorithm, we see that the algorithm will stop execution when the **if (A[i] == searchValue)** is evaluated to true
- Therefore the critical operation is this **if (A[i] == searchValue)** statement
- Also, the number of times the critical operation will be executed will depend on the actual input data (it may execute a few times for some input data and more/less than that for a different data)
- For this reason, we will be interested to count how many times the critical operation is executed in the **worst case** scenario
- The analysis of an algorithm based on the worst case scenario of the count is known as worst case scenario analysis

Sequential Search Running Time

- It is easy to see that the sequential search algorithm will execute its critical operation as many times as the size of the array
- Thus if we define a function **f(n)** where **n** is the size of the array and **f** is a function that tells us how many times the critical operation is executed in the worst case scenario, then we can easily see that

$$f(n) = n$$

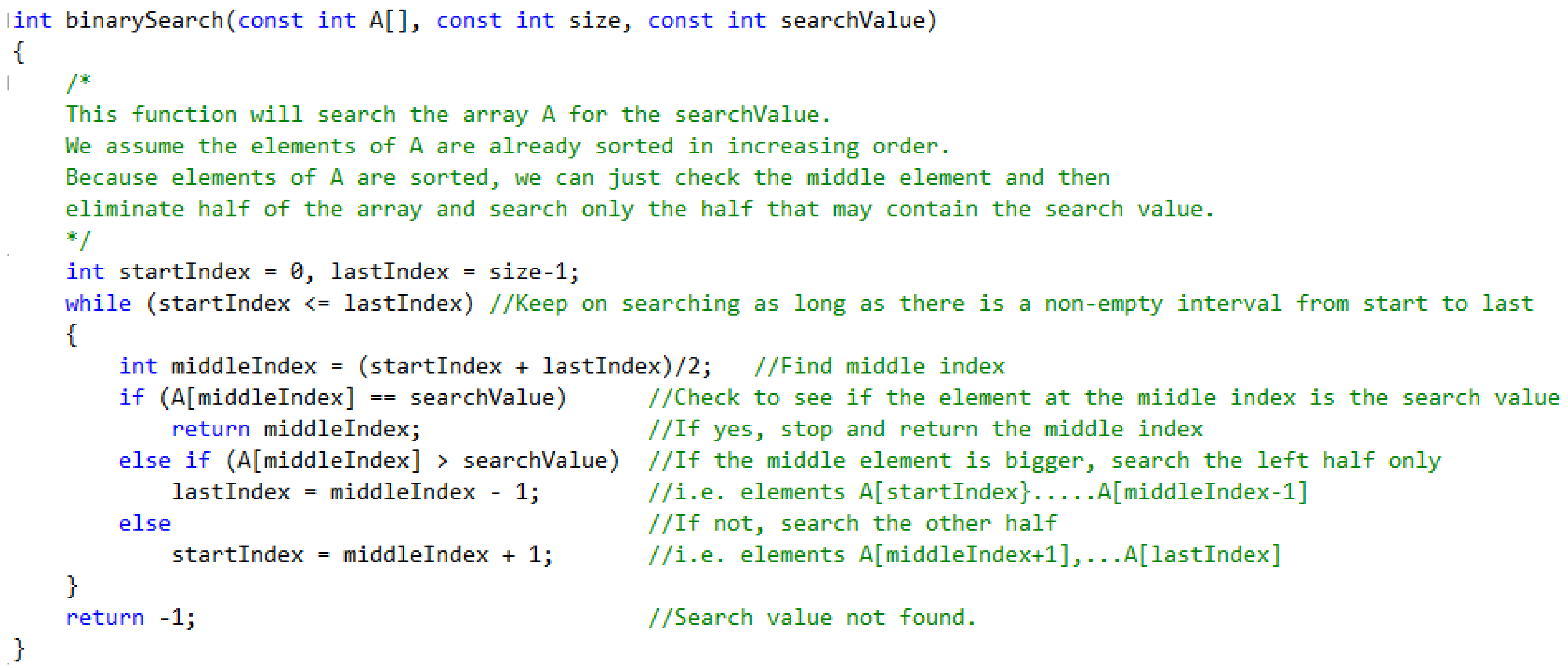
- We observe that the order of growth of the function **f** is linear
- Hence we conclude that the order of growth for the worst case running time of the sequential function is linear and write it as

$$O(n)$$

- For this reason, we say the sequential search is a linear algorithm

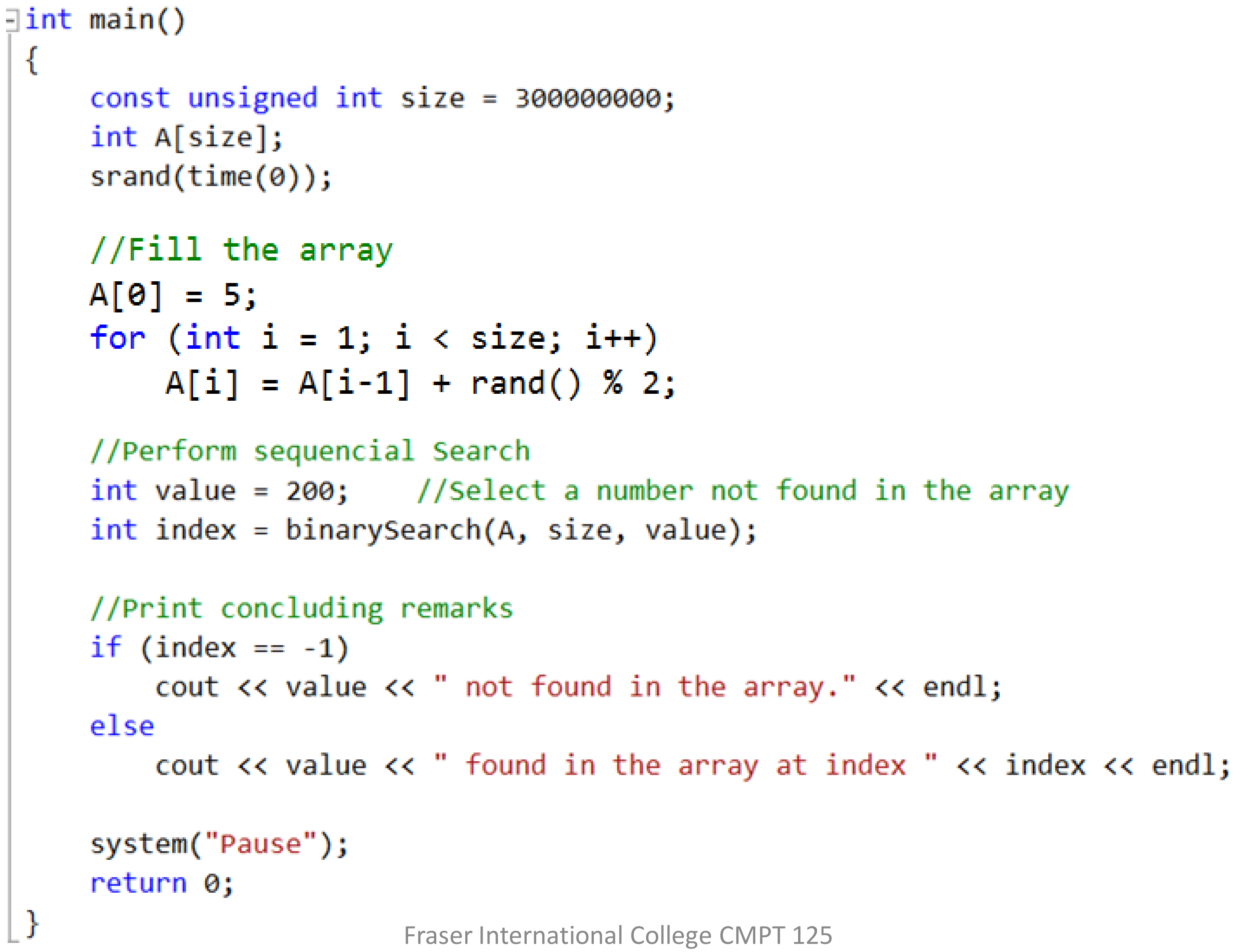
Binary Search

- Search a sorted array for a value by checking the middle element and then eliminating half of the array for the next search
- The following code demonstrates the idea:



```
int binarySearch(const int A[], const int size, const int searchValue)
{
    /*
    This function will search the array A for the searchValue.
    We assume the elements of A are already sorted in increasing order.
    Because elements of A are sorted, we can just check the middle element and then
    eliminate half of the array and search only the half that may contain the search value.
    */
    int startIndex = 0, lastIndex = size-1;
    while (startIndex <= lastIndex) //Keep on searching as long as there is a non-empty interval from start to last
    {
        int middleIndex = (startIndex + lastIndex)/2;   //Find middle index
        if (A[middleIndex] == searchValue)      //Check to see if the element at the miidle index is the search value
            return middleIndex;                 //If yes, stop and return the middle index
        else if (A[middleIndex] > searchValue)  //If the middle element is bigger, search the left half only
            lastIndex = middleIndex - 1;        //i.e. elements A[startIndex}.....A[middleIndex-1]
        else                                    //If not, search the other half
            startIndex = middleIndex + 1;       //i.e. elements A[middleIndex+1],...A[lastIndex]
    }
    return -1;                                  //Search value not found.
}
```

Binary Search



```cpp
int main()
{
    const unsigned int size = 300000000;
    int A[size];
    srand(time(0));

    //Fill the array
    A[0] = 5;
    for (int i = 1; i < size; i++)
        A[i] = A[i-1] + rand() % 2;

    //Perform sequencial Search
    int value = 200;    //Select a number not found in the array
    int index = binarySearch(A, size, value);

    //Print concluding remarks
    if (index == -1)
        cout << value << " not found in the array." << endl;
    else
        cout << value << " found in the array at index " << index << endl;

    system("Pause");
    return 0;
}
```

Binary Search Algorithm Running Time

- It is easy to see that **f(n) = 1 + f(n/2)**
- This mathematical equation is defined in a recursive manner
- That is to say that f is defined in terms of f
- We can also solve this recursive equation in order to define it in a closed form which gives

$$f(n) = 1 + \log_2 n \ldots \mathit{which\ is} \ldots O(\log n) \ldots \mathit{that\ is} \ldots\ \mathit{Logarithmic}$$

Sorting Algorithms

- Sorting is the process of arranging data in some order such as increasing order or decreasing order
- Several algorithms exist to sort data
- In this section, we will explore three different sorting algorithms known as selection sort algorithm, bubble sort algorithm, and insertion sort algorithm
- We will also analyze their worst case running times in order to compare them

Selection Sort Algorithm

- Given an array of data, this algorithm sorts given data as follow
  - Select the smallest element of the array and swap it with the element at index 0
  - At this point, the element at index 0 will be in its sorted position
  - Next, select the smallest element of the array excluding the element at index 0 and swap it with the element at index 1
  - At this point, the elements at indexes 0 and 1 will be in their sorted positions
  - Next, select the smallest element of the array excluding the elements at index 0 and at index 1 and swap it with the element at index 2
  - At this point, the elements at indexes 0, 1, and 2 will be in their sorted positions
  - Continue this process until all the elements of the array are sorted
- This algorithm is known as selection sort algorithm because at each stage, it selects the smallest element among all the unsorted elements and puts it in its correct position for sorting

Selection Sort Algorithm

```
void selectionSort(int A[], const int size)
{
        for (int i = 0; i < size; i++)
        {
                int index = i;
                for (int j = i+1; j < size; j++)
                        if (A[j] < A[index])
                                index = j;
                int temp = A[i];
                A[i] = A[index];
                A[index] = temp;
        }
}
int main()
{
        const unsigned int size = 300000000;
        int A[size];
        srand(time(0));
        for (int i = 0; i < size; i++)
                A[i] = rand();
        cout << "Original array ";
        printArray(A, size);
        selectionSort(A, size);
        cout << "Sorted array ";
        printArray(A, size);

        system("Pause");
        return 0;
}
```

Helper function

```
void printArray(const int A[], const int size)
{
    for (int i = 0; i < size; i++)
        cout << A[i] << "  ";
    cout << endl;
}
```

Selection Sort Algorithm Running Time

- In order to select the **k-th** smallest element, we see that we have to perform the comparison operation **(n-k)** times in the worst case
- Therefore, it is easy to see that the selection sort algorithm will require

  n-1 operations to select the smallest element

  n-2 operatiions to select the second smallest element

  n-3 operations to insert the third smallest element

  ⋮

  1 operation to select **(n-1)-th** smallest element

Selection Sort Algorithm Running Time

- Therefore

$$f(n) = n-1 + n-2 + n-3 + \ldots + 2 + 1 = n(n-1)/2$$

- That is

$$f(n) = 0.5n^2 - 0.5n$$

- Once again, in Big-O notation, we are concerned with the behaviour or order of the function **f(n)** as **n** becomes bigger and bigger
- Therefore we see that the f(n) for selection sort is simply quadratic
- We write it as **$f(n) = O(n^2)$ i.e. Quadratic**

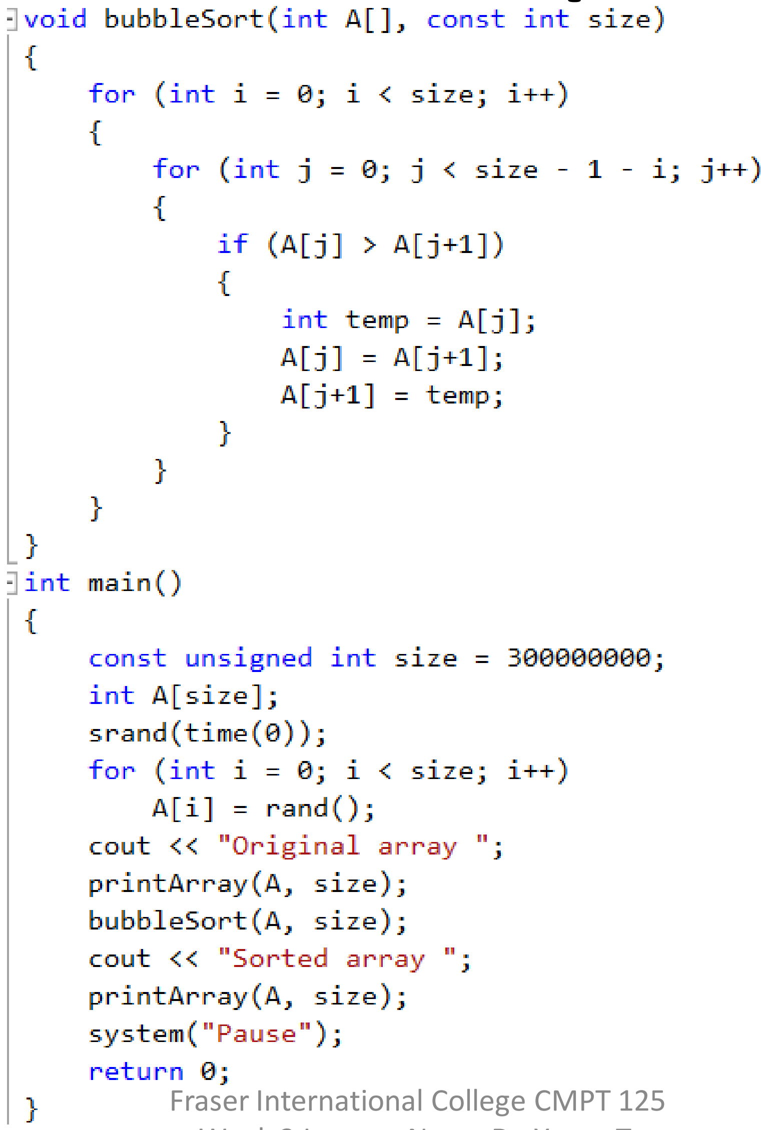
Sorting C++ Arrays Bubble Sort Algorithm

- In bubble sort, as its name implies, the idea is to bubble up elements upwards until all elements are sorted
- Consider the procedure for putting the largest element of an array to its right place:
  - Compare **A[0]** and **A[1]** and re-arrange them so that **A[1]** >= **A[0]**
  - Then compare **A[1]** and **A[2]** and re-arrange them so that **A[2]** >= **A[1]**
  - Then do the same for **A[2]** and **A[3]**
  - **⋮**
  - Finally do the same for **A[n-2]** and **A[n-1]**. Remember the last element has index **(n-1)**

Bubble Sort (Cont.)

- In one pass (that is **n-1** comparisons and swappings), the largest element will be bubbled up to its right position at **A[n-1]**
- In the next pass, the bubbling procedure is performed starting with **A[0]** and **A[1]** and going all the way to **A[n-3]** and **A[n-2]** to put the second largest element to its right place at **A[n-2]**
- And so on so forth... until all elements are sorted
- The following code does the job!

Bubble Sort (Cont.)



```
void bubbleSort(int A[], const int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size - 1 - i; j++)
        {
            if (A[j] > A[j+1])
            {
                int temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
int main()
{
    const unsigned int size = 300000000;
    int A[size];
    srand(time(0));
    for (int i = 0; i < size; i++)
        A[i] = rand();
    cout << "Original array ";
    printArray(A, size);
    bubbleSort(A, size);
    cout << "Sorted array ";
    printArray(A, size);
    system("Pause");
    return 0;
}
```

Bubble Sort Algorithm Running Time

- It is easy to see that the upper loop will iterate for **n** times
- The inner loop will iterate

  n-1 times when **i = 0**

  n-2 times when **i = 1**

  n-3 times when **i = 2**

  ⋮

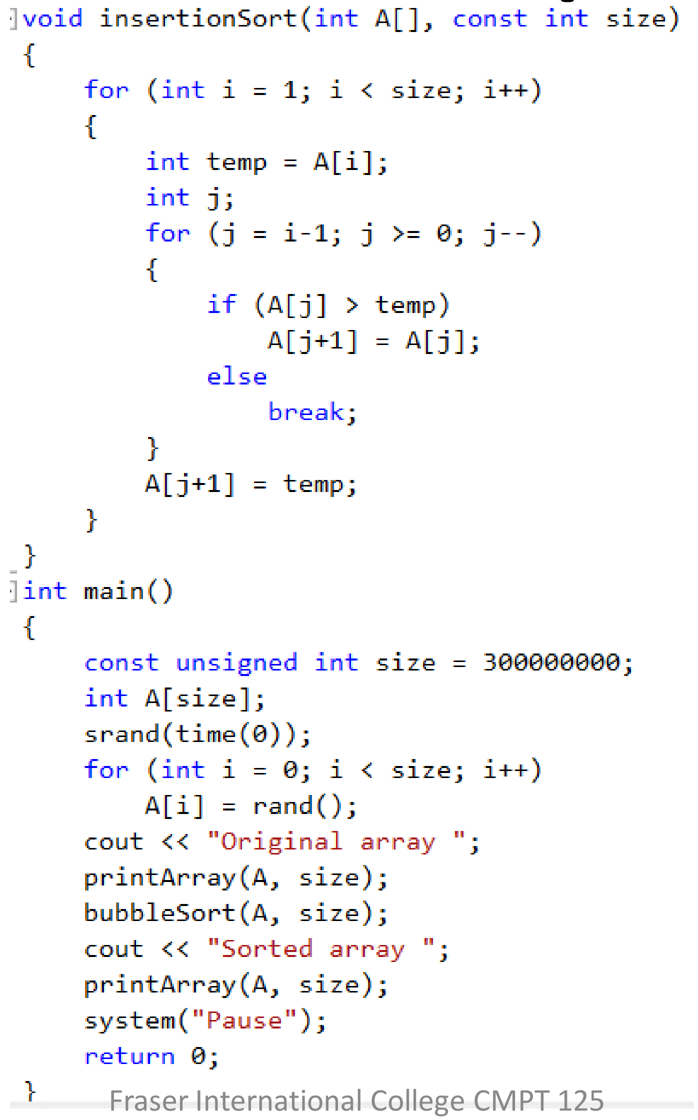
  1 time when **i = n-2**
- Summing up, we get **f(n) = 0.5n^2-0.5n**
- In Big-O notation, this is written as

  $f(n)=O(n^2)$

Insertion Sort

- Yet another algorithm to sort arrays is the insertion sort
- In this algorithm, assuming we are sorting in increasing order,
  - We first assume the first element is already in its right place
  - Then pick the second element and insert it either before or after the first element depending if it is less than or greater than the first element
  - Next, pick the third element and insert it either before the first element or between the first and second elements or after the second element
  - Generally, pick element at index **k** and insert it in its right place between indexes **0** and **k**
  - After inserting the last element, the array will be sorted
- See the following implementation

Insertion Sort (cont.)



```
void insertionSort(int A[], const int size)
{
    for (int i = 1; i < size; i++)
    {
        int temp = A[i];
        int j;
        for (j = i-1; j >= 0; j--)
        {
            if (A[j] > temp)
                A[j+1] = A[j];
            else
                break;
        }
        A[j+1] = temp;
    }
}
int main()
{
    const unsigned int size = 300000000;
    int A[size];
    srand(time(0));
    for (int i = 0; i < size; i++)
        A[i] = rand();
    cout << "Original array ";
    printArray(A, size);
    bubbleSort(A, size);
    cout << "Sorted array ";
    printArray(A, size);
    system("Pause");
    return 0;
}
```

Insertion Sort Algorithm Running Time

- It is easy to see that the upper loop will iterate n-1 times; where n is the number of elements in the array
- In the first pass, the critical operation (that is the **if-statement**) will be executed once
- In the second pass, it will be executed twice in the worst case scenario
- In the third pass trice, etc
- In total, it will be executed 1+2+3+…+(n-1) which shows

$$\mathbf{f(n) = 0.5n^2-0.5n}$$

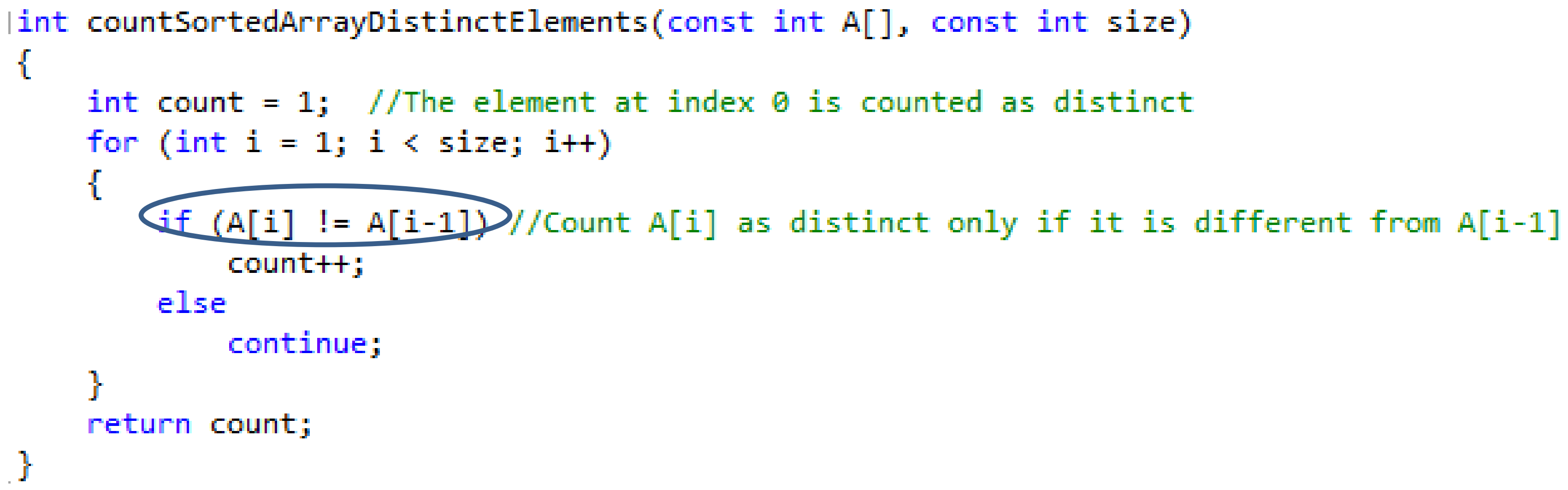
- Therefore we conclude that the insertion sort is **$O(n^2)$**

Counting Critical Operations

- In order to count critical operations of any given procedure (that is an implementation of an algorithm); we need to carefully count the operation of interest (example “if statement”) inside loops and other code segments.
- In particular, we look for some mathematical relationship between our count and the size of the object under consideration
- The following example demonstrates the idea

Counting Critical Operations

- **Example:-** Given the following procedure to count the distinct elements of a sorted array; how many times does the critical operation, shown inside an elliptical curve, get executed?



```
int countSortedArrayDistinctElements(const int A[], const int size)
{
    int count = 1;  //The element at index 0 is counted as distinct
    for (int i = 1; i < size; i++)
    {
        if (A[i] != A[i-1]) //Count A[i] as distinct only if it is different from A[i-1]
            count++;
        else
            continue;
    }
    return count;
}
```

Counting Critical Operations

- **Example:- Now, consider a similar procedure but this time** to count the distinct elements of any array; how many times does the critical operation, shown inside an elliptical curve, get executed?

```
int countArrayDistinctElements(const int A[], const int size)
{
    int count = 1;  //The element at index 0 is counted as distinct
    for (int i = 1; i < size; i++)
    {
        bool found = false; //Check if A[i] is found among A[0], A[1],...,A[i-1]
        for (int j = 0; j < i; j++)
        {
            if (A[j] == A[i])
            {
                found = true;
                break;
            }
            else
                continue;
        }
        if (found ==  false)    //If A[i] was not found, count it as distinct
            count++;
    }
    return count;
}
```

Complexity of Algorithms Summary

Algorithm	Sequential Search	Binary Search	Selection Sort	Bubble Sort	Insertion Sort	Counting Distinct (sorted array)	Counting Distinct (any array)
Complexity	O(n)	O(log n)	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(n)	$O(n^2)$
	Linear	Logarithmic	Quadratic	Quadratic	Quadratic	Linear	Quadratic