Reference variables, C++ vectors and C++ recursive functions

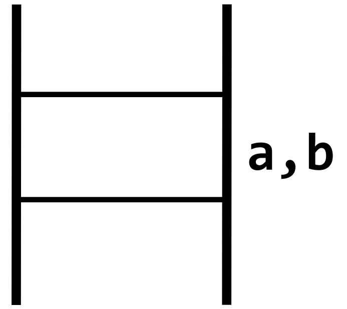
In this week

- Reference Variables
- Parameter Passing by Reference
- C++ vectors
- Recursion and C++ Recursive Functions
- Asserting Arguments
- Divide and Conquer Algorithms
- The Quick Sort Algorithm
- Complexity of the Quick Sort Algorithm

Reference Variables

- In C++, a reference variable is a variable name that is a second name (or an alias) to another same data type variable name
- Example

```
int a;//an int data type variable
int &b = a;//a reference (alias) to a
```

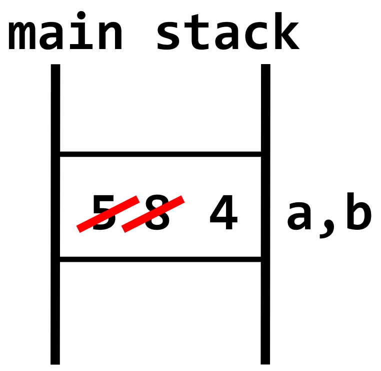


- Now both the variables **a** and **b** refer to the same memory space

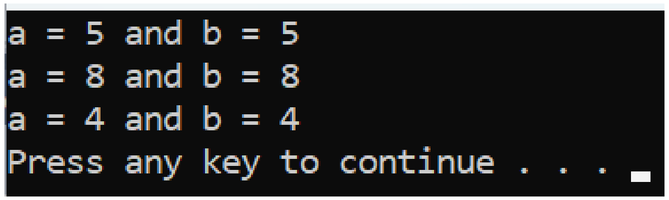
Reference Variables

- We can initialize the memory space using either variable **a** or **b**, modify a value in the memory space using either of the variables **a** or **b**, or use a value in the memory space using either of the variables **a** or **b**

```
int main()
{
    int a; //An int data type variable
    int &b = a; //A reference to a
    a = 5;
    cout << "a = " << a << " and b = " << b << endl;
    b = b + 3;
    cout << "a = " << a << " and b = " << b << endl;
    a = b - 4;
    cout << "a = " << a << " and b = " << b << endl;
    system("Pause");
    return 0;
}
```



OUTPUT

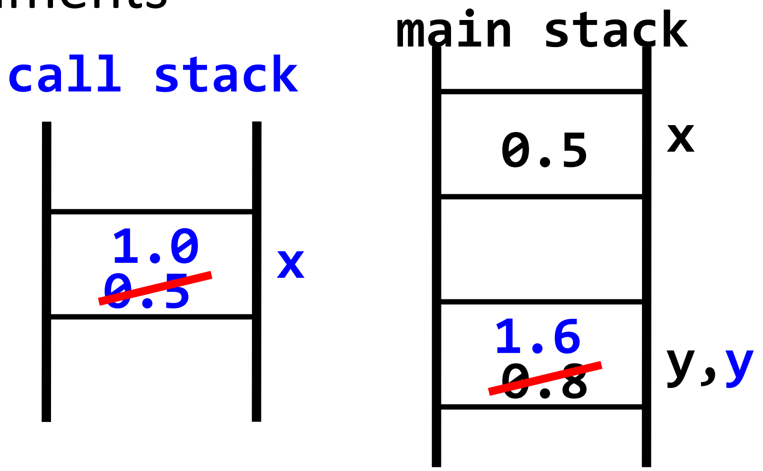


```
a = 5 and b = 5
a = 8 and b = 8
a = 4 and b = 4
Press any key to continue . . .
```

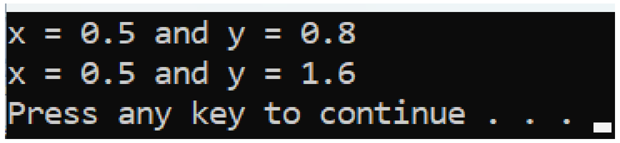
Parameter Passing by Reference

- Some or all of the parameters of a C++ function can be made reference variables to their corresponding arguments
- In this case any modification to the parameters will also modify their corresponding arguments

```
void foo(double x, double& y)
{
    //x is a copy of its argument x
    //y is a reference to its argument y
    x = 2*x; //This will not modify the argument x
    y = 2*y; //This will modify the argument y
}
int main()
{
    double x = 0.5, y = 0.8;
    cout << "x = " << x << " and y = " << y << endl;
    foo(x, y);
    cout << "x = " << x << " and y = " << y << endl;

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



OUTPUT



```
x = 0.5 and y = 0.8
x = 0.5 and y = 1.6
Press any key to continue . . .
```

Parameter Passing by Reference

- When a parameter of a function is made a reference to its argument, then we say the function is using parameter passing by reference
- Thus in the previous example, the function is using parameter passing by value for its parameter x and it is using parameter passing by reference for its parameter y
- Parameter passing by reference has **three main advantages**
  - ➢ **It makes function call faster because no data is copied into the call stack memory**
  - ➢ **It helps reduce the memory spaces used in a program**
  - ➢ **It makes it possible to modify an argument to a function inside the function**

Parameter Passing by Reference

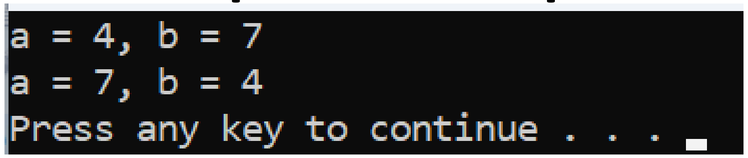
- **Practice Exercise:-** Define a void C++ function named **swap** that takes two integer arguments and swaps its arguments. Use the following test program to test your function.

answer

```cpp
void swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

```cpp
int main()
{
    srand(time(0));
    int a = rand() % 10, b = rand() % 20;
    cout << "a = " << a << ", b = " << b << endl;
    swap(a, b);
    cout << "a = " << a << ", b = " << b << endl;
    system("Pause");
    return 0;
}
```

Sample Run Output



```
a = 4, b = 7
a = 7, b = 4
Press any key to continue . . .
```

C++ vectors

- A C++ vector is similar to an array with the ability to resize itself automatically when an element is inserted or erased (i.e. removed)
- Similar to arrays, elements of a C++ vector are accessed with indexes
- In order to use C++ vectors,

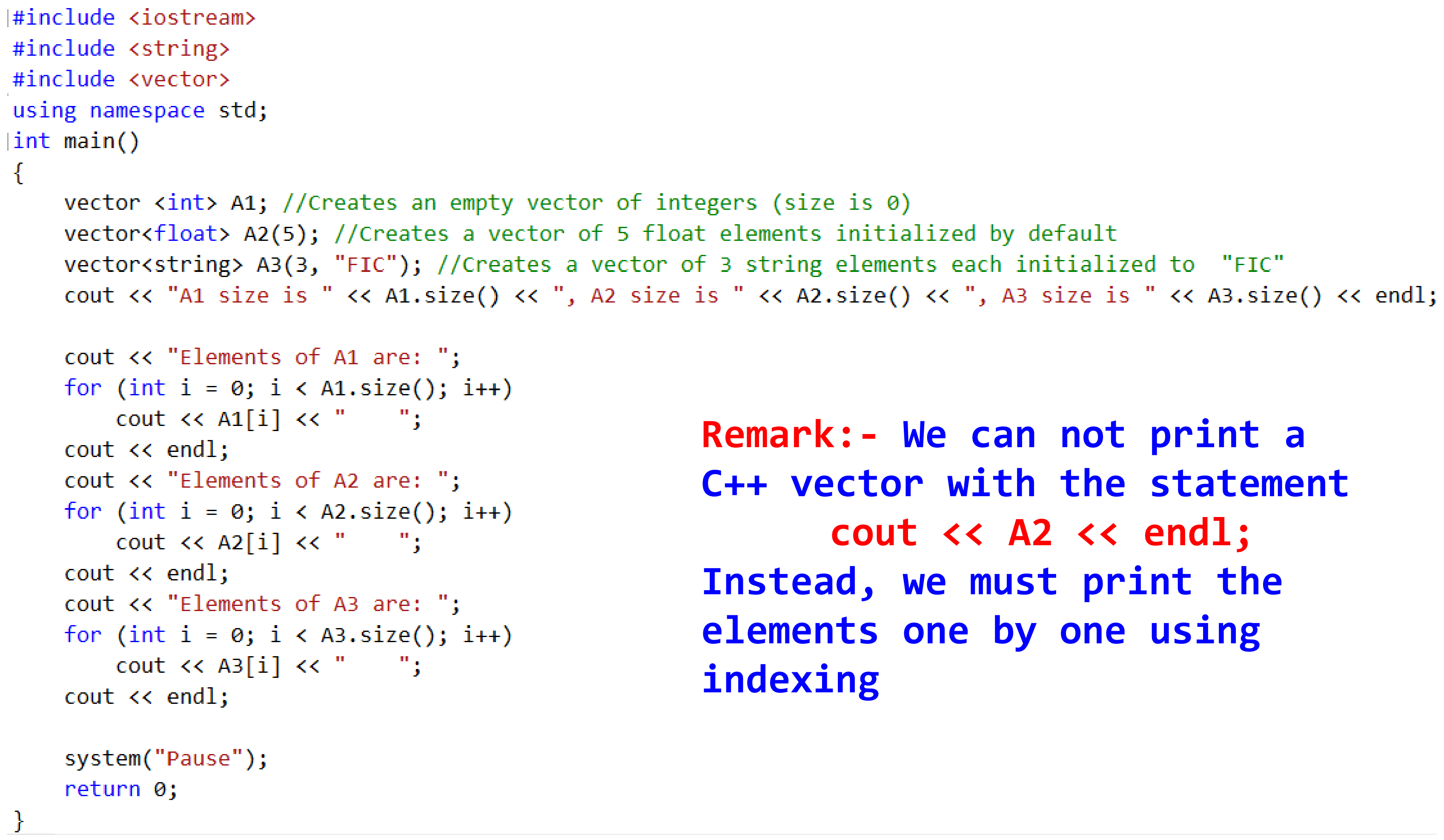
  #include <vector>

  include directive is needed

C++ vectors

- C++ vectors support the following methods and operations
  - ✓ **creation**: create an empty vector, a vector filled with some elements, or a vector copied from another vector
  - ✓ **size**: returns the number of elements in a vector
  - ✓ **indexing**: index starts at 0 and increments by 1
  - ✓ **push_back**: appends a value to the vector as a last element and increment size by 1
  - ✓ **empty**: test if the vector is empty (that is if size is zero). Returns true if the vector is empty.
  - ✓ **erase**: remove one or more values from the vector and reduce the size accordingly
  - ✓ **insert**: insert one or more values into the vector and increment the size accordingly

Creation and Traversal of C++ vectors



```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main()
{
    vector <int> A1; //Creates an empty vector of integers (size is 0)
    vector<float> A2(5); //Creates a vector of 5 float elements initialized by default
    vector<string> A3(3, "FIC"); //Creates a vector of 3 string elements each initialized to  "FIC"
    cout << "A1 size is " << A1.size() << ", A2 size is " << A2.size() << ", A3 size is " << A3.size() << endl;

    cout << "Elements of A1 are: ";
    for (int i = 0; i < A1.size(); i++)
        cout << A1[i] << "    ";
    cout << endl;
    cout << "Elements of A2 are: ";
    for (int i = 0; i < A2.size(); i++)
        cout << A2[i] << "    ";
    cout << endl;
    cout << "Elements of A3 are: ";
    for (int i = 0; i < A3.size(); i++)
        cout << A3[i] << "    ";
    cout << endl;

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

Remark:- We can not print a C++ vector with the statement

`cout << A2 << endl;`

Instead, we must print the elements one by one using indexing

Appending elements to C++ vectors and Copying C++ vectors

- We can create an empty vector and then append as many elements as we wish using the **push_back** method
- We can also create a vector by **copying** from another vector of the **same data type**
- When a vector is created by copying from another vector, then the new vector will store a copy of the elements of the given vector in its own memory space (i.e. separate from the memory space belonging to the vector being copied)
- This means, after copying, the two vectors will be independent of each other
- Thus modifying any one of them does not modify the other
- See the following example…

Appending elements to C++ vectors and Copying C++ vectors

```
int main()
{
    //Create an empty vector and append few elements to it
    vector <int> v1;
    for (int i = 0; i < 5; i++)
        v1.push_back(2*i+1);
    cout << "v1 size is " << v1.size() << endl;
    cout << "Elements of v1 are: ";
    for (int i = 0; i < v1.size(); i++)
        cout << v1[i] << "    ";
    cout << endl;
    //Create a vector copied from v1
    vector<int> v2(v1);  //Alternatively, we could write it as vector<int> v2 = v1;
    cout << "v2 size is " << v2.size() << endl;
    cout << "Elements of v2 are: ";
    for (int i = 0; i < v2.size(); i++)
        cout << v2[i] << "    ";
    cout << endl;
    //Append some more elements to v2
    v2.push_back(-6);
    v2.push_back(0);
    cout << "v1 size is still " << v1.size() << " but v2 size is now " << v2.size() << endl;
    cout << "Elements of v1 are: ";
    for (int i = 0; i < v1.size(); i++)
        cout << v1[i] << "    ";
    cout << endl;
    cout << "Elements of v2 are: ";
    for (int i = 0; i < v2.size(); i++)
        cout << v2[i] << "    ";
    cout << endl;
    system("Pause");
    return 0;
 }
```

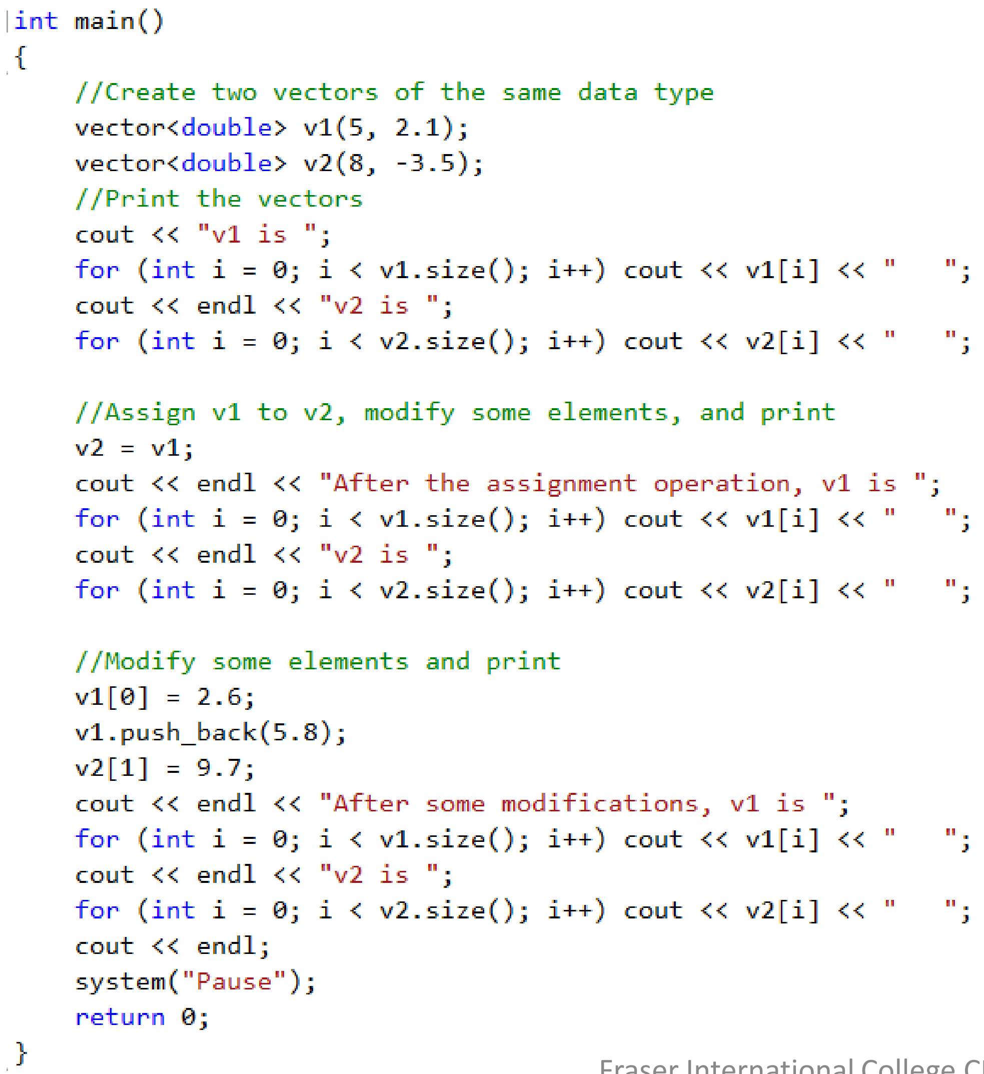
Assignment operator for C++ vectors

- Given two C++ vectors v1 and v2 of the **same data type** having some elements, consider the assignment operator

```
        v2 = v1;
```

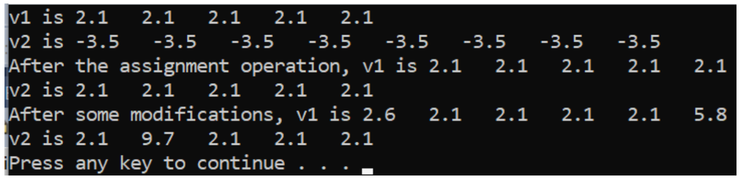
- The assignment operator copies the elements of v1 to v2
- In this case, the original elements of v2 are deleted and replaced by the elements of v1
- Importantly, after the assignment operation v2 will have its own memory space for its elements separate from v1
- Therefore v1 and v2 will work independent of each other
- This means modifying any one of them does not modify the other
- Thus we conclude that C++ vector variables work in the same way as simple data type variables such as int, float, string, double, bool, etc. See the following example

Assignment operator for C++ vectors



```cpp
int main()
{
    //Create two vectors of the same data type
    vector<double> v1(5, 2.1);
    vector<double> v2(8, -3.5);
    //Print the vectors
    cout << "v1 is ";
    for (int i = 0; i < v1.size(); i++) cout << v1[i] << "    ";
    cout << endl << "v2 is ";
    for (int i = 0; i < v2.size(); i++) cout << v2[i] << "    ";

    //Assign v1 to v2, modify some elements, and print
    v2 = v1;
    cout << endl << "After the assignment operation, v1 is ";
    for (int i = 0; i < v1.size(); i++) cout << v1[i] << "    ";
    cout << endl << "v2 is ";
    for (int i = 0; i < v2.size(); i++) cout << v2[i] << "    ";

    //Modify some elements and print
    v1[0] = 2.6;
    v1.push_back(5.8);
    v2[1] = 9.7;
    cout << endl << "After some modifications, v1 is ";
    for (int i = 0; i < v1.size(); i++) cout << v1[i] << "    ";
    cout << endl << "v2 is ";
    for (int i = 0; i < v2.size(); i++) cout << v2[i] << "    ";
    cout << endl;
    system("Pause");
    return 0;
}
```

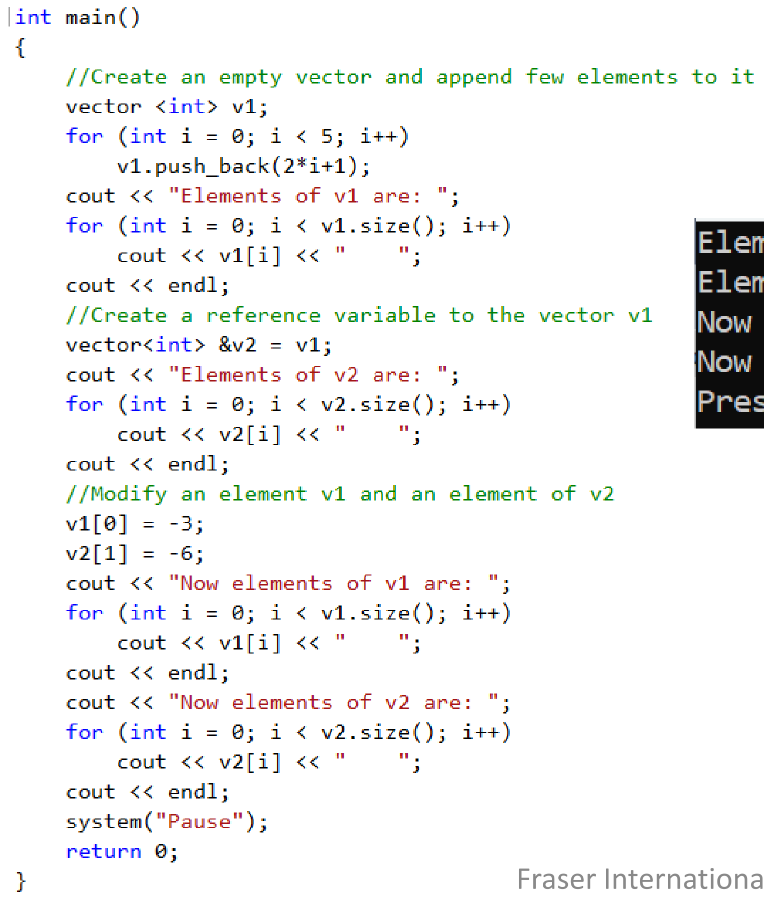
OUTPUT



```
v1 is 2.1    2.1    2.1    2.1    2.1
v2 is -3.5    -3.5    -3.5    -3.5    -3.5    -3.5    -3.5    -3.5
After the assignment operation, v1 is 2.1    2.1    2.1    2.1    2.1
v2 is 2.1    2.1    2.1    2.1    2.1
After some modifications, v1 is 2.6    2.1    2.1    2.1    2.1    5.8
v2 is 2.1    9.7    2.1    2.1    2.1
Press any key to continue . . .
```

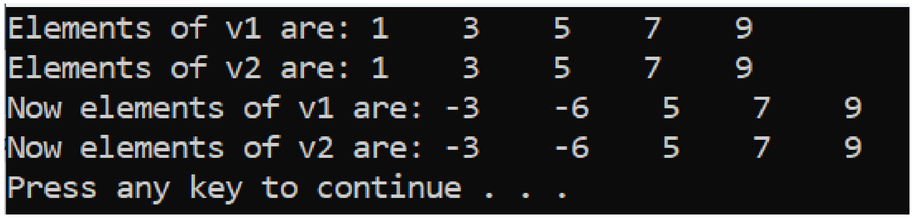
References to vectors

- We may also create a reference variable to a vector in which case the reference variable acts as an alias (second name) to the original vector and modifying any one of the vectors will also modify the other



```
int main()
{
    //Create an empty vector and append few elements to it
    vector <int> v1;
    for (int i = 0; i < 5; i++)
        v1.push_back(2*i+1);
    cout << "Elements of v1 are: ";
    for (int i = 0; i < v1.size(); i++)
        cout << v1[i] << "    ";
    cout << endl;
    //Create a reference variable to the vector v1
    vector<int> &v2 = v1;
    cout << "Elements of v2 are: ";
    for (int i = 0; i < v2.size(); i++)
        cout << v2[i] << "    ";
    cout << endl;
    //Modify an element v1 and an element of v2
    v1[0] = -3;
    v2[1] = -6;
    cout << "Now elements of v1 are: ";
    for (int i = 0; i < v1.size(); i++)
        cout << v1[i] << "    ";
    cout << endl;
    cout << "Now elements of v2 are: ";
    for (int i = 0; i < v2.size(); i++)
        cout << v2[i] << "    ";
    cout << endl;
    system("Pause");
    return 0;
}
```

OUTPUT

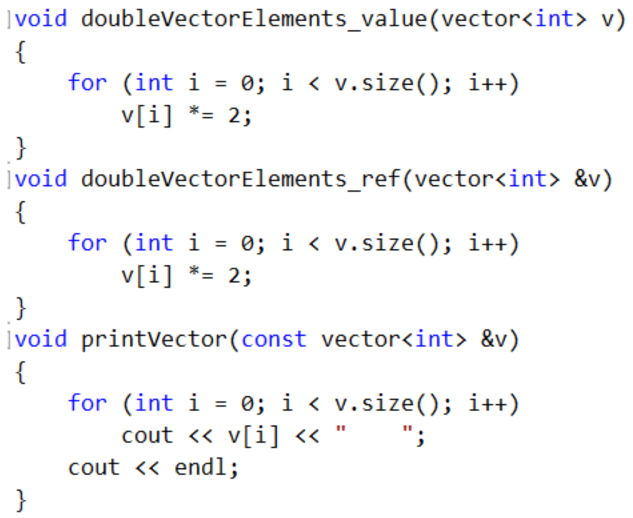


```
Elements of v1 are: 1    3    5    7    9
Elements of v2 are: 1    3    5    7    9
Now elements of v1 are: -3    -6    5    7    9
Now elements of v2 are: -3    -6    5    7    9
Press any key to continue . . .
```

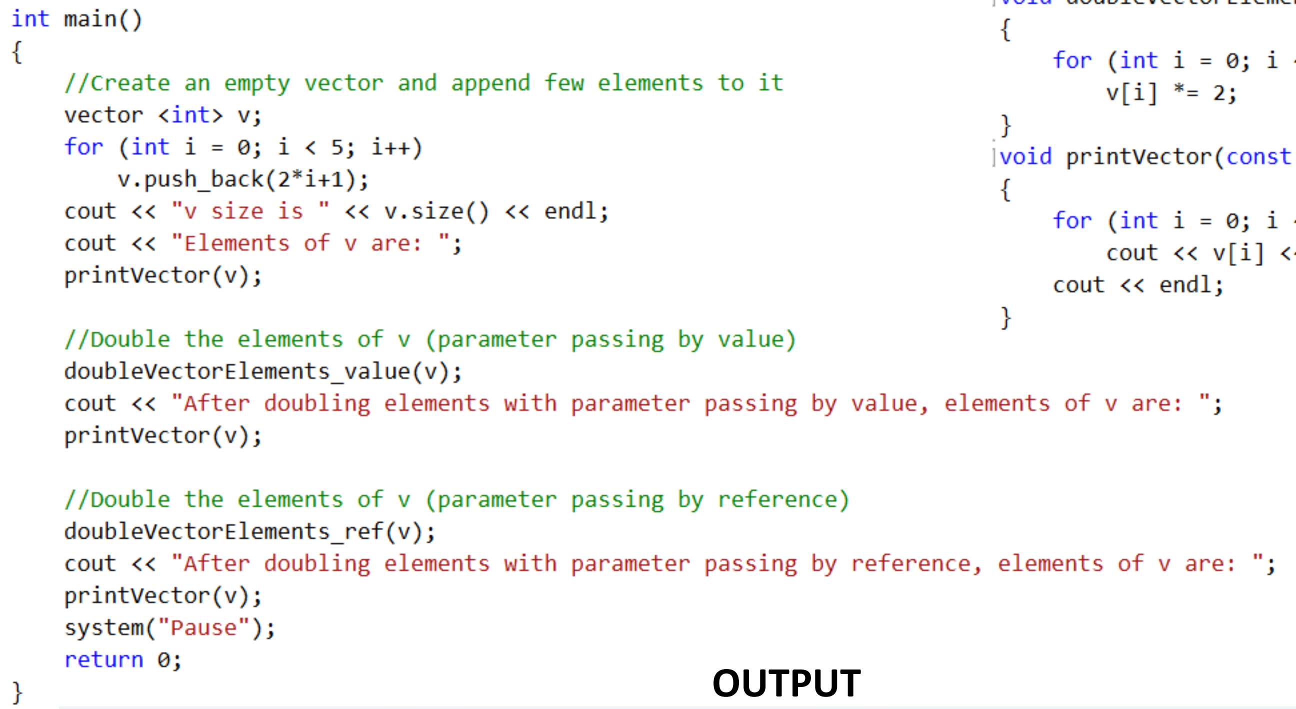
Passing C++ vectors to functions

- We can pass a vector to a function parameter of the **same data type** either using parameter passing by value or by reference
- Passing a vector to a function by value will copy the vector argument to the function parameter hence modifying the parameter does NOT modify the argument
- Passing a vector to a function by reference makes a the function parameter a reference to the argument hence modifying the parameter will also modify the argument

Passing C++ vectors to functions

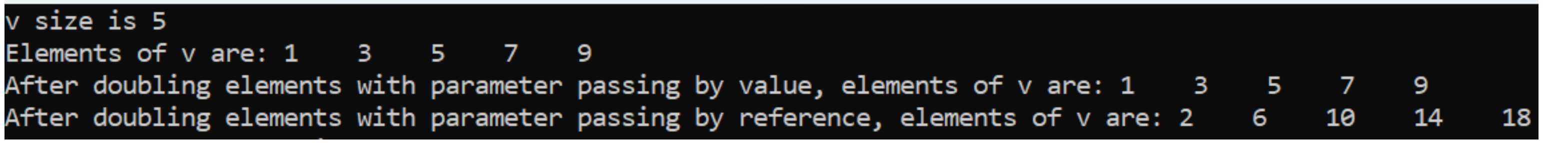


```cpp
void doubleVectorElements_value(vector<int> v)
{
    for (int i = 0; i < v.size(); i++)
        v[i] *= 2;
}
void doubleVectorElements_ref(vector<int> &v)
{
    for (int i = 0; i < v.size(); i++)
        v[i] *= 2;
}
void printVector(const vector<int> &v)
{
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << "    ";
    cout << endl;
}
```



```cpp
int main()
{
    //Create an empty vector and append few elements to it
    vector <int> v;
    for (int i = 0; i < 5; i++)
        v.push_back(2*i+1);
    cout << "v size is " << v.size() << endl;
    cout << "Elements of v are: ";
    printVector(v);

    //Double the elements of v (parameter passing by value)
    doubleVectorElements_value(v);
    cout << "After doubling elements with parameter passing by value, elements of v are: ";
    printVector(v);

    //Double the elements of v (parameter passing by reference)
    doubleVectorElements_ref(v);
    cout << "After doubling elements with parameter passing by reference, elements of v are: ";
    printVector(v);
    system("Pause");
    return 0;
}
```

OUTPUT



```
v size is 5
Elements of v are: 1    3    5    7    9
After doubling elements with parameter passing by value, elements of v are: 1    3    5    7    9
After doubling elements with parameter passing by reference, elements of v are: 2    6    10    14    18
```

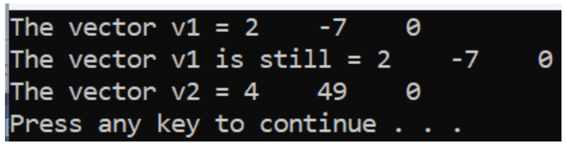
Returning C++ vectors from functions

- We can also return a C++ vector from a function in the same way as we return a simple data type value from a function
- In this case, a copy of the vector will be returned from the function
- See the following example

```cpp
#include <iostream>
#include <vector>
using namespace std;
vector<int> getSquares(const vector<int> &x)
{
    vector<int> ans;
    for (int i = 0; i < x.size(); i++)
        ans.push_back(x[i]*x[i]);
    return ans;
}
void printVector(const vector<int> &v)
{
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << "    ";
    cout << endl;
}
```

```cpp
int main()
{
    vector<int> v1;
    v1.push_back(2);
    v1.push_back(-7);
    v1.push_back(0);
    cout << "The vector v1 = ";
    printVector(v1);

    //Create the squares of v1 vector
    vector<int> v2;
    v2 = getSquares(v1);
    cout << "The vector v1 is still = ";
    printVector(v1);
    cout << "The vector v2 = ";
    printVector(v2);
    system("Pause");
    return 0;
}
```

OUTPUT



```
The vector v1 = 2    -7    0
The vector v1 is still = 2    -7    0
The vector v2 = 4    49    0
Press any key to continue . . .
```

Inserting and erasing elements

- We may also insert a new element into a vector or erase an element from a vector at a specific valid index as follows
- In order to insert an element into an index `i`, we proceed as

```
v.insert(v.begin() + i, value_to_insert);
```

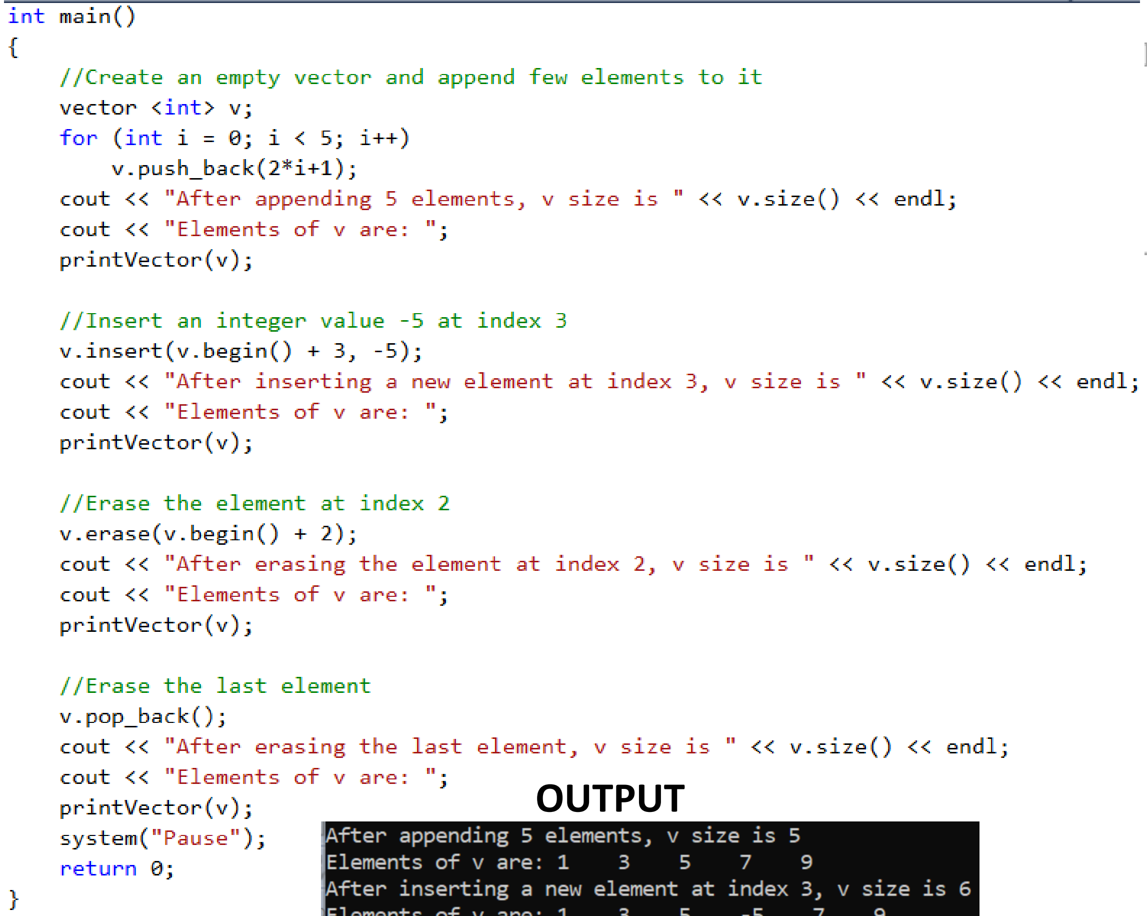
- In this case, the index `i` must satisfy the condition that `i >= 0 && i <= v.size()`
- When `i = v.size()` then the element is appended at the end end of the vector just like push_back
- In order to erase (i.e. remove) an element at index `i`, we proceed as

```
v.erase(v.begin() + i);
```

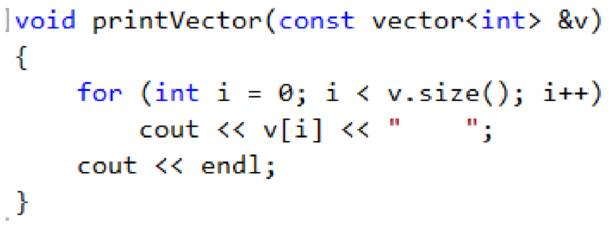
- In this case, the index `i` must satisfy the condition that `i >= 0 && i < v.size()`
- We may also erase the last element in a vector using

```
v.pop_back();
```

Inserting and erasing elements

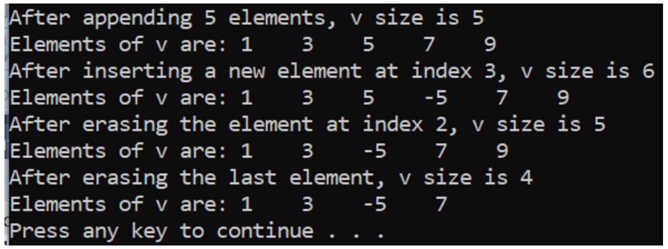


```cpp
int main()
{
    //Create an empty vector and append few elements to it
    vector <int> v;
    for (int i = 0; i < 5; i++)
        v.push_back(2*i+1);
    cout << "After appending 5 elements, v size is " << v.size() << endl;
    cout << "Elements of v are: ";
    printVector(v);

    //Insert an integer value -5 at index 3
    v.insert(v.begin() + 3, -5);
    cout << "After inserting a new element at index 3, v size is " << v.size() << endl;
    cout << "Elements of v are: ";
    printVector(v);

    //Erase the element at index 2
    v.erase(v.begin() + 2);
    cout << "After erasing the element at index 2, v size is " << v.size() << endl;
    cout << "Elements of v are: ";
    printVector(v);

    //Erase the last element
    v.pop_back();
    cout << "After erasing the last element, v size is " << v.size() << endl;
    cout << "Elements of v are: ";
    printVector(v);
    system("Pause");
    return 0;
}
```



```cpp
void printVector(const vector<int> &v)
{
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << "    ";
    cout << endl;
}
```

OUTPUT



```
After appending 5 elements, v size is 5
Elements of v are: 1    3    5    7    9
After inserting a new element at index 3, v size is 6
Elements of v are: 1    3    5    -5    7    9
After erasing the element at index 2, v size is 5
Elements of v are: 1    3    -5    7    9
After erasing the last element, v size is 4
Elements of v are: 1    3    -5    7
Press any key to continue . . .
```

Recursion and Recursive Functions

- Consider the mathematical ***factorial*** function ***f*** defined for non-negative integers as follows

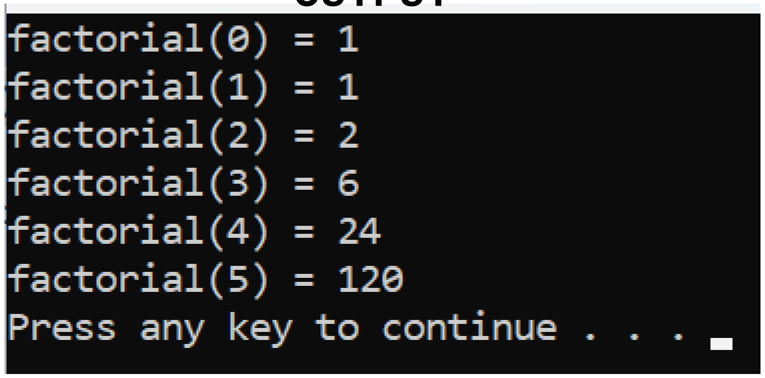
```
    f(n) = n*(n-1)*(n-2)*...*2*1   if n > 0
                      and
    f(0) = 1   (by definition)
```

- Given the value of **n**, we can easily calculate the value of ***f(n)*** as follows: ***f(0)=1*** (definition), ***f(1)=1***, ***f(2)=2*1=2***, ***f(3)=3*2*1=6***, ***f(4)=4*3*2*1=24***, etc
- We may also define a C++ function to compute the factorial a given argument as follows

Recursion and Recursive Functions

```cpp
int factorial(const int n)
{
    //Pre-condition: n >= 0
    //Post-condition: n factorial is returned
     if (n == 0)
         return 1;
     else
     {
         int f = 1;
         for (int i = n; i > 0; i--)
             f *= i;
         return f;
     }
}
int main()
{
     for (int k = 0; k < 6; k++)
         cout << "factorial(" << k << ") = " << factorial(k) << endl;
     system("Pause");
     return 0;
}
```

OUTPUT



```
factorial(0) = 1
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
Press any key to continue . . .
```

Recursion and Recursive Functions

- Another way to define the mathematical ***factorial*** function ***f*** is as follows

```
f(n) = n * f(n-1)   if n > 0
            and
f(0) = 1    (by definition)
```

- Using this function, we can calculate for example

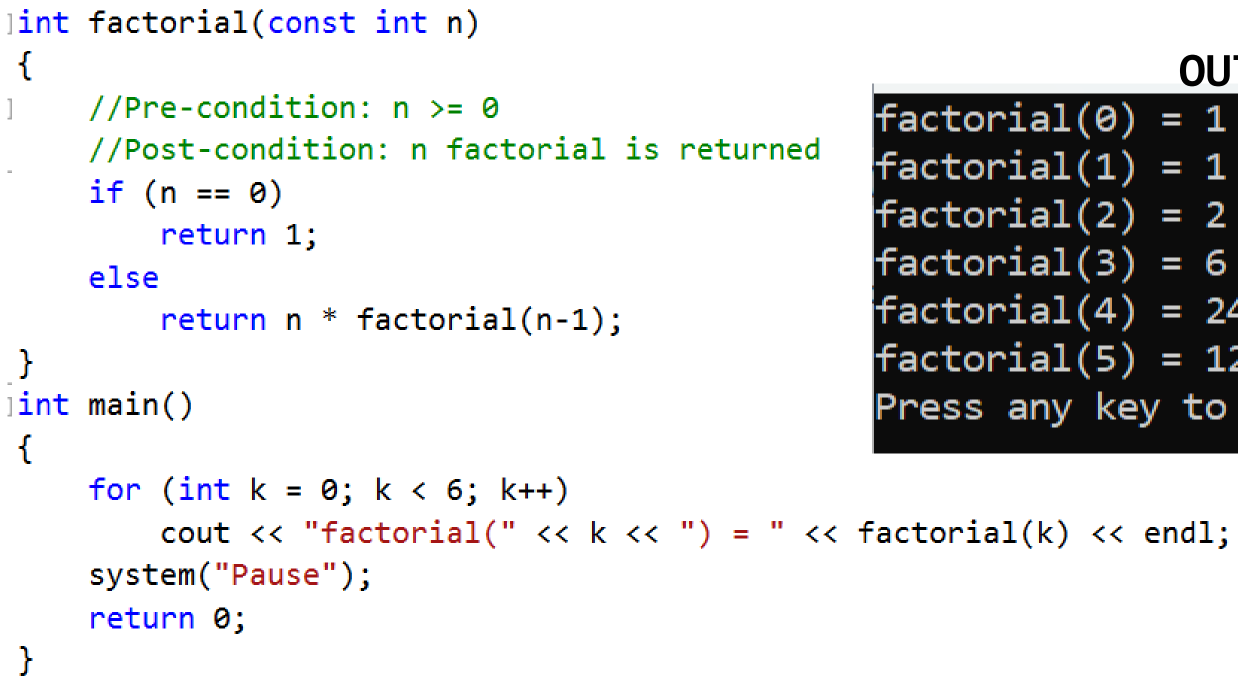
```
f(4) = 4 * f(3)
     = 4 * 3 * f(2)
     = 4 * 3 * 2 * f(1)
     = 4 * 3 * 2 * 1 * f(0)
     = 4 * 3 * 2 * 1 * 1
     = 24
```

Recursion and Recursive Functions

- The definition ***f(n) = n * f(n-1)*** defines the factorial function ***f*** using the function ***f*** itself and is known as **recursive function definition**
- The special case ***f(0) = 1*** which does not define the function ***f*** in terms of itself is known as the **base case**
- The case ***f(n) = n * f(n-1)*** is known as the **recursive step**
- For any recursive function definition to be valid, it must have a base case and repeated application of the recursive step must approach the base case; otherwise the process will never end
- Recursion is the process of solving a problem using recursive definition

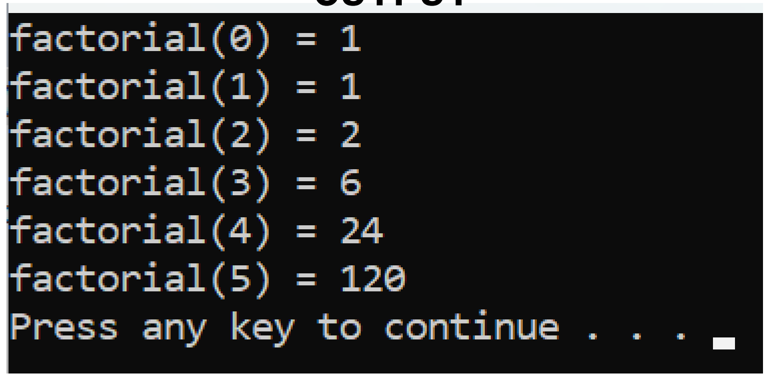
C++ Recursive Functions

- We may also define a C++ function that implements the recursive definition of the factorial function thanks to the fact that C++ allows a function to call itself as shown below



```
int factorial(const int n)
{
    //Pre-condition: n >= 0
    //Post-condition: n factorial is returned
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
int main()
{
    for (int k = 0; k < 6; k++)
        cout << "factorial(" << k << ") = " << factorial(k) << endl;
    system("Pause");
    return 0;
}
```

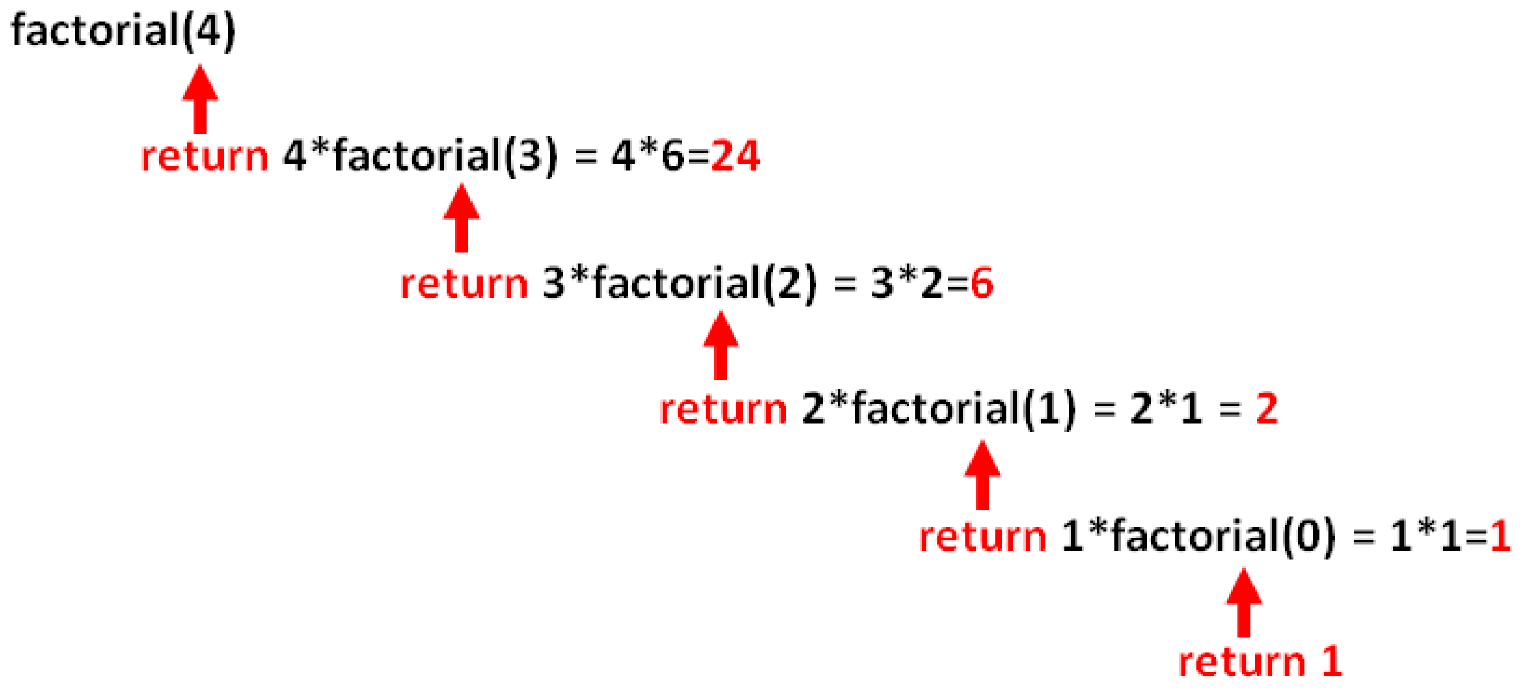
OUTPUT



```
factorial(0) = 1
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
Press any key to continue . . .
```

C++ Recursive Functions

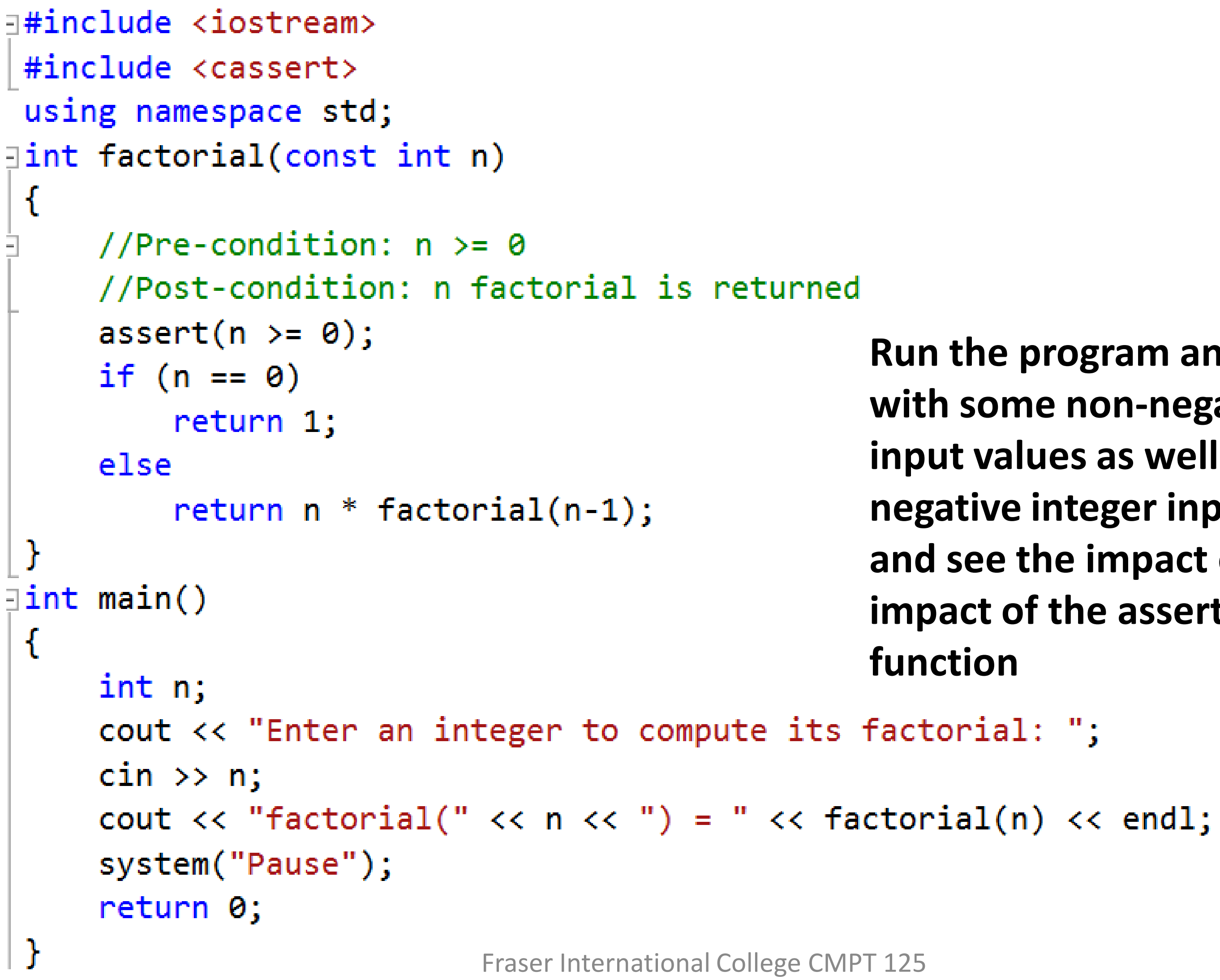
- Pictorial representation of the C++ recursive function definition for, say factorial(4), looks like as follows


factorial(4)
return 4*factorial(3) = 4*6=24
return 3*factorial(2) = 3*2=6
return 2*factorial(1) = 2*1 = 2
return 1*factorial(0) = 1*1=1
return 1

C++ Recursive Functions

- Given a non-negative integer argument, the function computes its factorial by repeatedly calling itself until the base case is reached
- The argument must be non-negative; otherwise an infinite recursion runtime error will occur
- One way to avoid infinite recursion runtime errors is to **assert** the argument to the function is non-negative
- C++ provides a built-in function named **assert** for this purpose which raises a runtime error when the assertion fails and will automatically **crash** the program
- In order to use the assert built-in function, we need to include the **cassert** library

C++ Recursive Functions



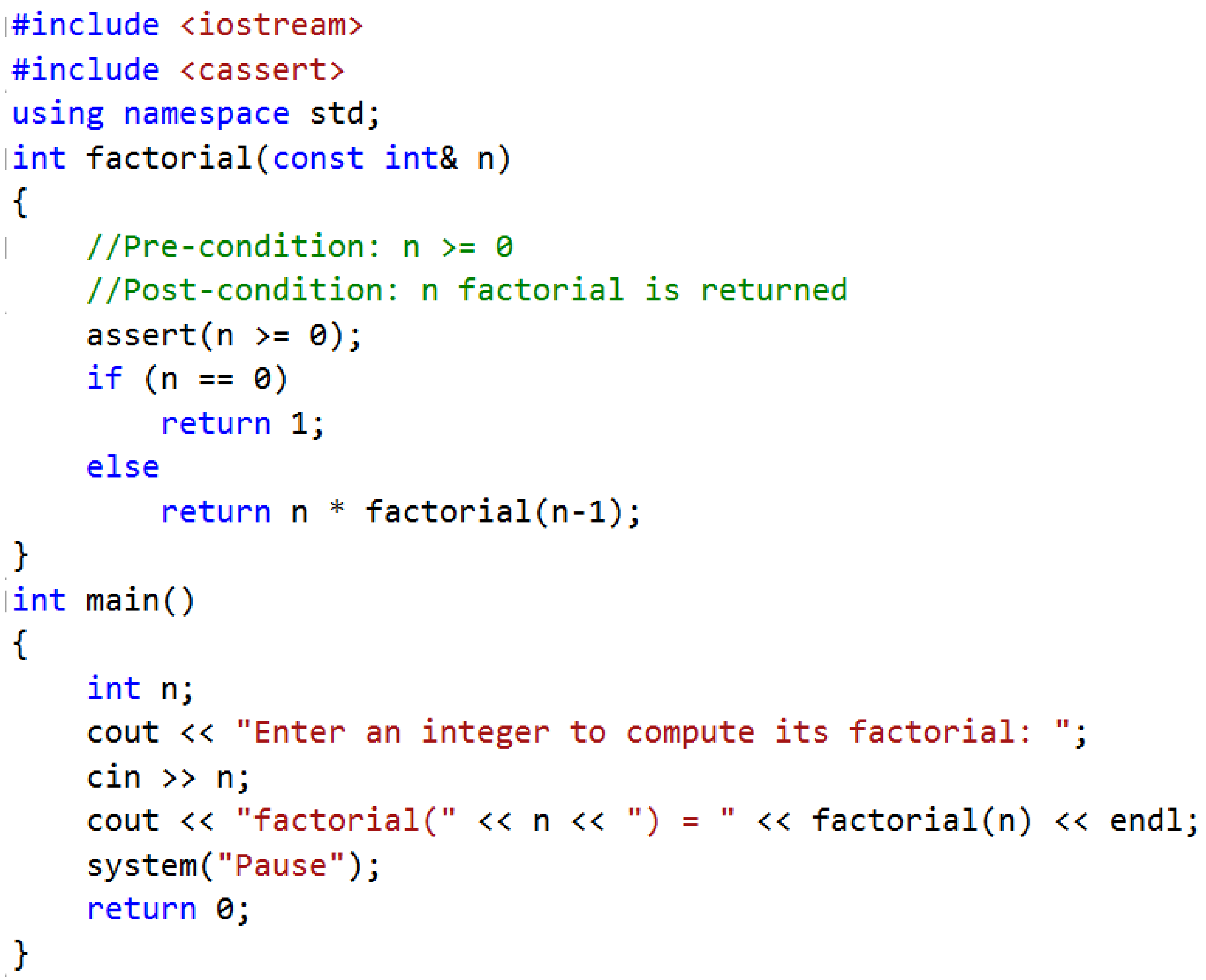
```
#include <iostream>
#include <cassert>
using namespace std;
int factorial(const int n)
{
    //Pre-condition: n >= 0
    //Post-condition: n factorial is returned
    assert(n >= 0);
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
int main()
{
    int n;
    cout << "Enter an integer to compute its factorial: ";
    cin >> n;
    cout << "factorial(" << n << ") = " << factorial(n) << endl;
    system("Pause");
    return 0;
}
```

Run the program and test it with some non-negative input values as well as some negative integer input values and see the impact of the impact of the assert built-in function

C++ Recursive Functions

- It should be noted that each recursive function call uses its own call stack memory
- This means as we call the function again and again, then more and more memory spaces will be used by our program
- In order to conserve memory spaces and to make the repeated function calls faster, we may use parameter passing by reference in the function definition
- Moreover we may use the constant qualifier to emphasize the fact that the function is not supposed to modify its argument…See below

C++ Recursive Functions



```
#include <iostream>
#include <cassert>
using namespace std;
int factorial(const int& n)
{
    //Pre-condition: n >= 0
    //Post-condition: n factorial is returned
    assert(n >= 0);
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
int main()
{
    int n;
    cout << "Enter an integer to compute its factorial: ";
    cin >> n;
    cout << "factorial(" << n << ") = " << factorial(n) << endl;
    system("Pause");
    return 0;
}
```

C++ Recursive Functions

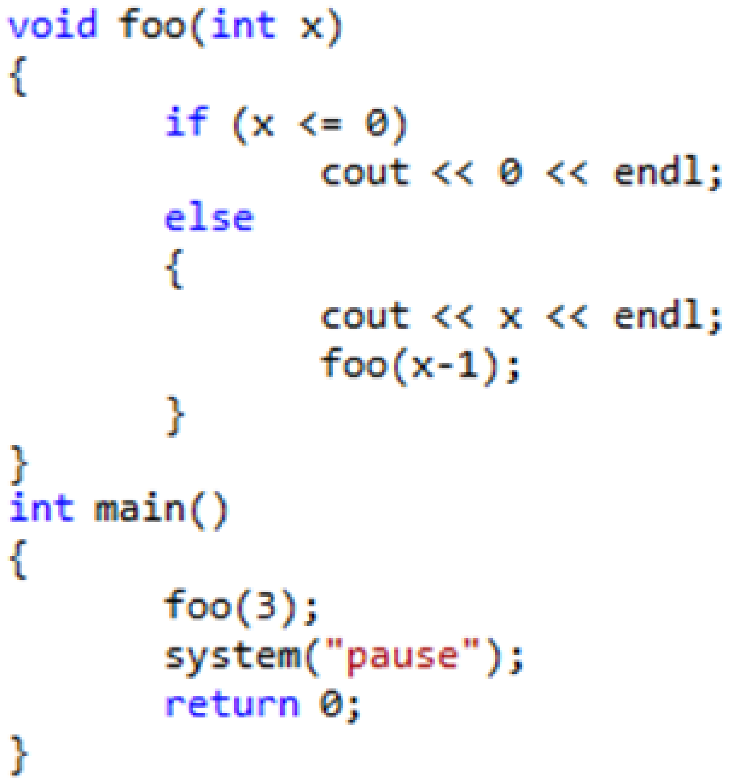
- The heart of recursion in programming lies in the fact that a function, such as the **factorial** function shown above, was able to call itself
- That is to say each function call knows where it was called from and thus returns to the same place from where it was called from
- In C++ recursive function calls are managed using a special memory called the stack memory that keeps track of function calls in a very elegant way as described below

C++ Recursive Functions

- The stack memory is a data structure that allows addition of new items on top of it and only one item at a time can be extracted from the top of the stack
- Whenever a C++ function is called in a program, the function call is added on top of the stack
- Processing of a series of function calls is performed by adding the function calls on top of the stack in the order they were called and then processing the function on top of the stack
- Once processed a function is cleared from the stack and then the next function on top of the stack is processed
- When we first call a function from the main program, we start with an empty stack and whenever we return to the main program the stack memory becomes empty

C++ Recursive Functions Examples

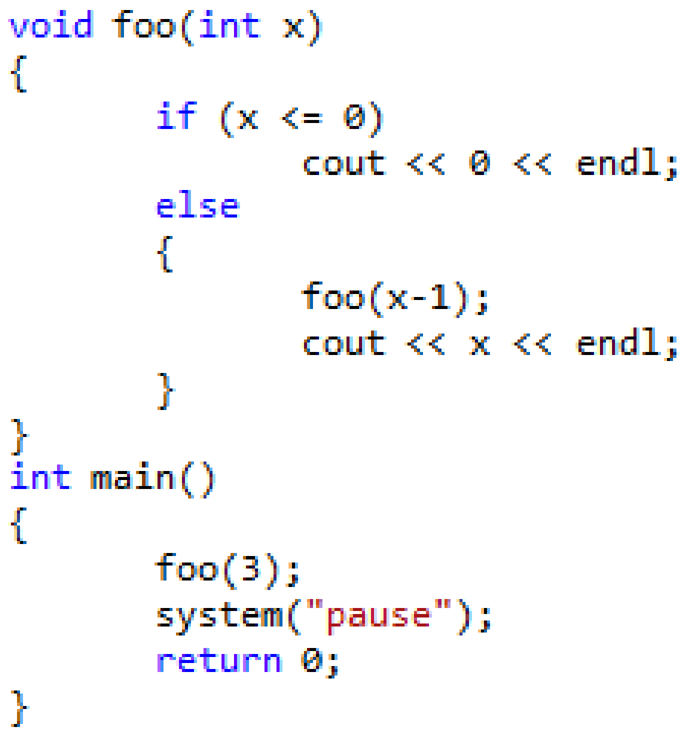
- Now we look at several examples and analyze these examples in order to reinforce our understanding of C++ recursive functions. Analyze the following recursive function and a test program and determine its output



```
void foo(int x)
{
        if (x <= 0)
                cout << 0 << endl;
        else
        {
                cout << x << endl;
                foo(x-1);
        }
}
int main()
{
        foo(3);
        system("pause");
        return 0;
}
```

C++ Recursive Functions Examples

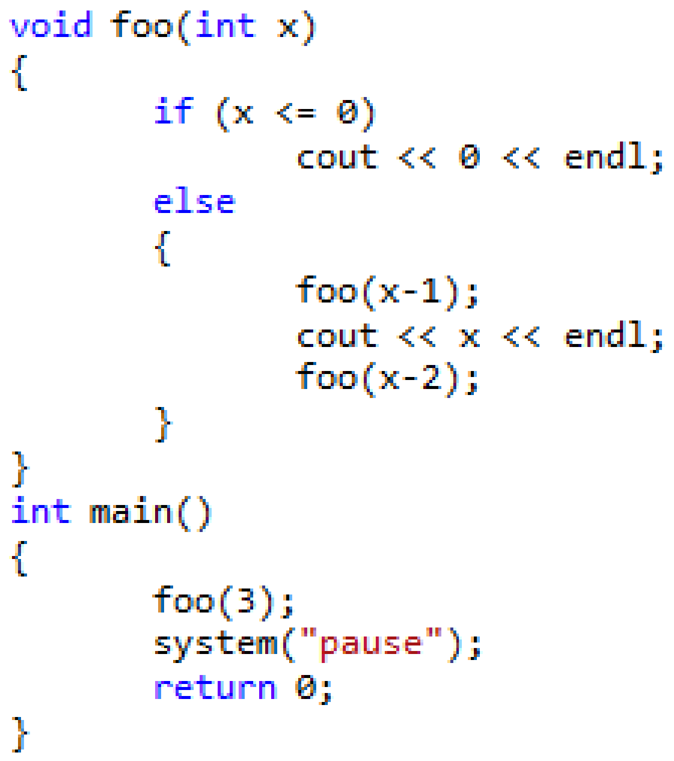
- Analyze the following recursive function and a test program and determine its output



```
void foo(int x)
{
        if (x <= 0)
                cout << 0 << endl;
        else
        {
                foo(x-1);
                cout << x << endl;
        }
}
int main()
{
        foo(3);
        system("pause");
        return 0;
}
```

C++ Recursive Functions Examples

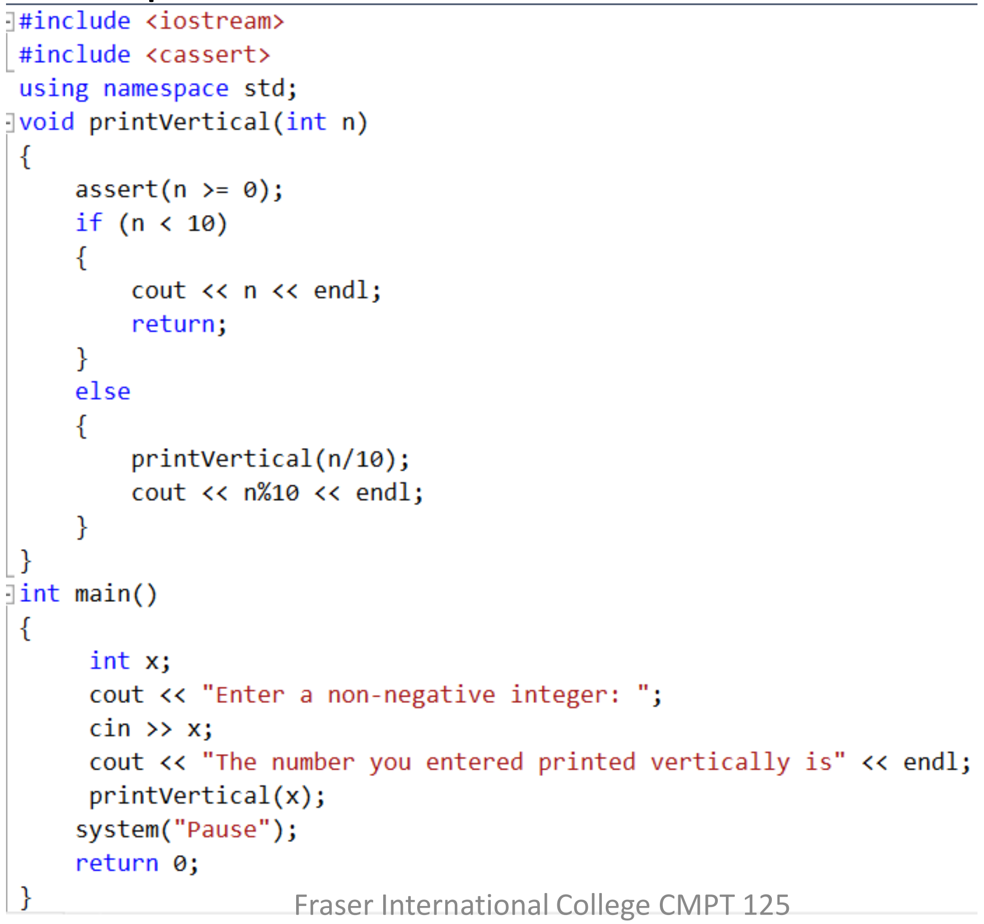
- Analyze the following recursive function and a test program and determine its output



```
void foo(int x)
{
        if (x <= 0)
                cout << 0 << endl;
        else
        {
                foo(x-1);
                cout << x << endl;
                foo(x-2);
        }
}
int main()
{
        foo(3);
        system("pause");
        return 0;
}
```

C++ Recursive Functions Examples

- Analyze the following recursive function and a test program and determine its output. Assume x = 1782



```cpp
#include <iostream>
#include <cassert>
using namespace std;
void printVertical(int n)
{
    assert(n >= 0);
    if (n < 10)
    {
        cout << n << endl;
        return;
    }
    else
    {
        printVertical(n/10);
        cout << n%10 << endl;
    }
}
int main()
{
     int x;
     cout << "Enter a non-negative integer: ";
     cin >> x;
     cout << "The number you entered printed vertically is" << endl;
     printVertical(x);
    system("Pause");
    return 0;
}
```

Recursive Search Algorithms

- Searching is a very common process in computing
- While searching can easily be performed using non-recursive algorithms, it is a problem that has a natural recursive solution as well
- For the sake of exploring recursion and how to think recursively, we now look at sequential and binary recursive search algorithms

Sequential Search

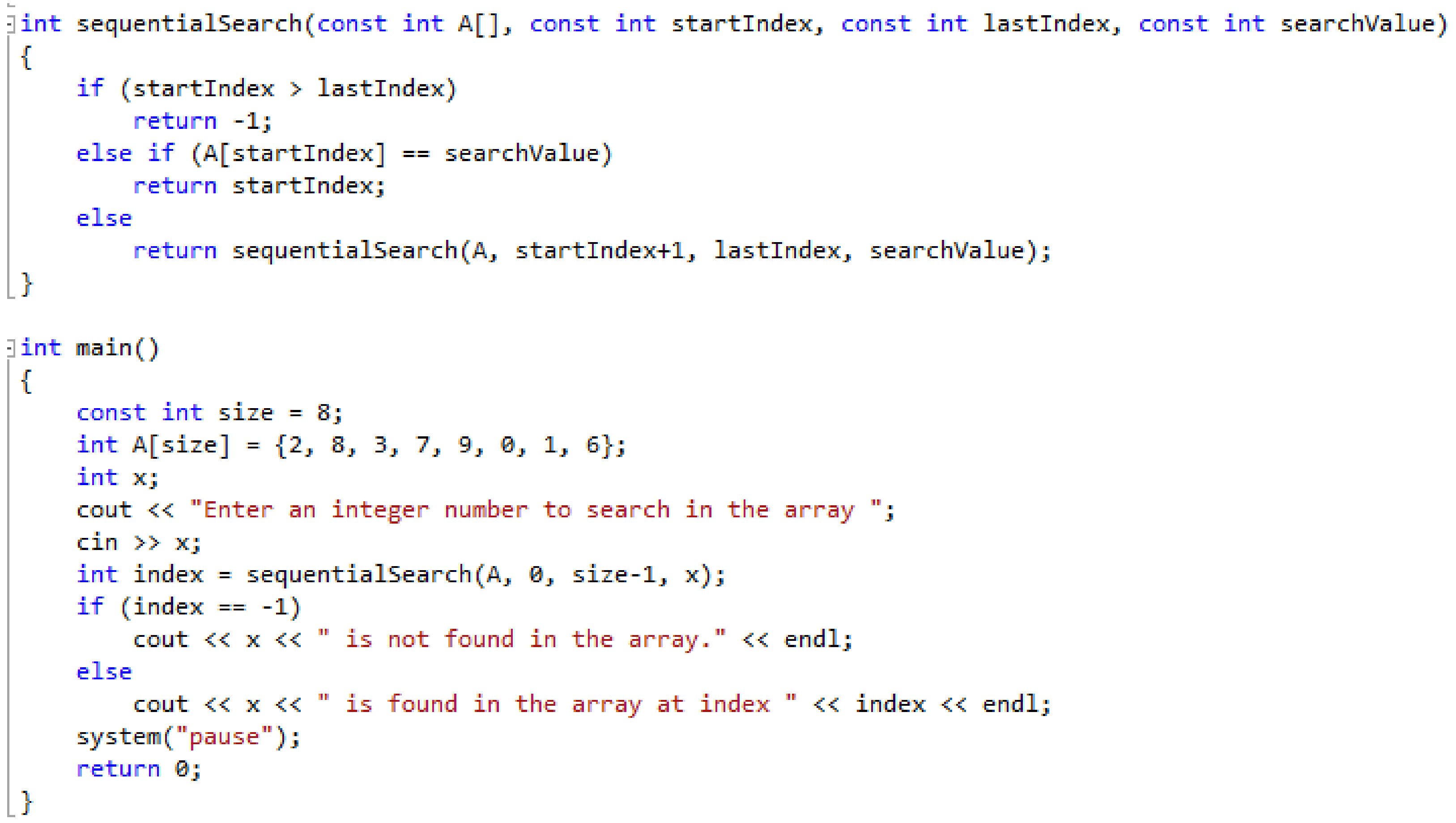
- **Problem Statement**: Given an array or a vector of **n** elements (say integers), we would like to search the array/vector if a search value is found in the array/vector. If the search value is found, we would like to return the index of the element matching the search value; otherwise we would like to return -1 to mean the search value is not found in the array/vector.
- Let us name the array/vector **A** and the value to be searched **searchValue**

Sequential Search

- **Solution**: A recursive solution can be given as follows: Check if the **searchValue** is equal to the first element of the array/vector. If yes, we return the index of the first element; otherwise we search in the remaining elements of the array/vector. We stop the search when there are no more elements to search in the array/vector
- **Programming**: In order to keep track of the first element of the array/vector we need to have the knowledge of the **start index** at each step. Moreover, in order not to exceed the last element we will also need to have the knowledge of the **last index** at each step

Sequential Search

- Thus the following function does the job

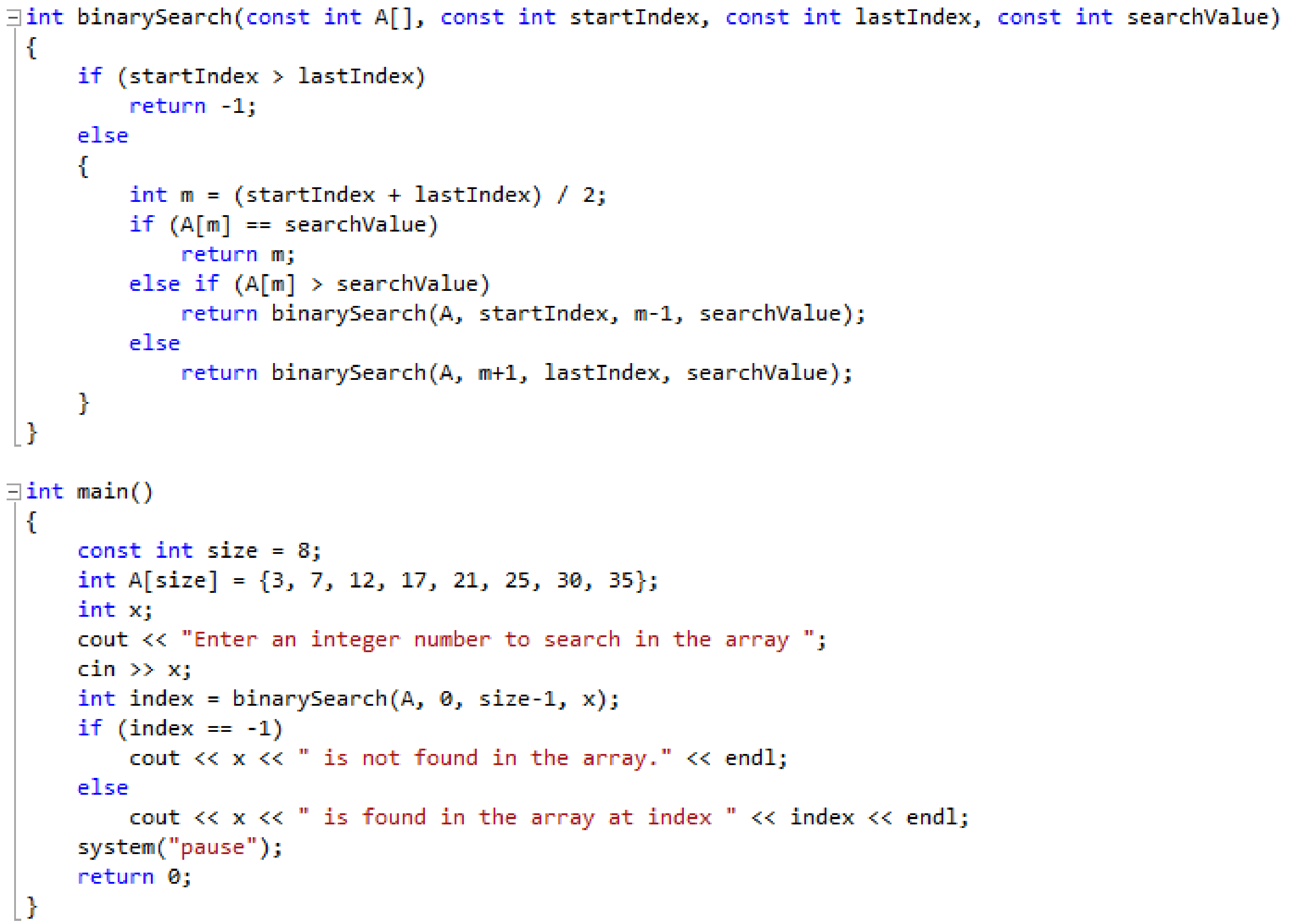


```
int sequentialSearch(const int A[], const int startIndex, const int lastIndex, const int searchValue)
{
    if (startIndex > lastIndex)
        return -1;
    else if (A[startIndex] == searchValue)
        return startIndex;
    else
        return sequentialSearch(A, startIndex+1, lastIndex, searchValue);
}

int main()
{
    const int size = 8;
    int A[size] = {2, 8, 3, 7, 9, 0, 1, 6};
    int x;
    cout << "Enter an integer number to search in the array ";
    cin >> x;
    int index = sequentialSearch(A, 0, size-1, x);
    if (index == -1)
        cout << x << " is not found in the array." << endl;
    else
        cout << x << " is found in the array at index " << index << endl;
    system("pause");
    return 0;
}
```

Binary Search

- In binary search algorithm, we assume that the array/vector is already sorted (say in increasing order) and we don't have to search sequentially
- Instead, we check the middle element of the array/vector if it is equal to the **searchValue**; if yes we return the middle index; otherwise we search one half of the array/vector depending which side the **searchValue** lies
- The following function implements the recursive binary search algorithm

Binary Search



```cpp
int binarySearch(const int A[], const int startIndex, const int lastIndex, const int searchValue)
{
    if (startIndex > lastIndex)
        return -1;
    else
    {
        int m = (startIndex + lastIndex) / 2;
        if (A[m] == searchValue)
            return m;
        else if (A[m] > searchValue)
            return binarySearch(A, startIndex, m-1, searchValue);
        else
            return binarySearch(A, m+1, lastIndex, searchValue);
    }
}

int main()
{
    const int size = 8;
    int A[size] = {3, 7, 12, 17, 21, 25, 30, 35};
    int x;
    cout << "Enter an integer number to search in the array ";
    cin >> x;
    int index = binarySearch(A, 0, size-1, x);
    if (index == -1)
        cout << x << " is not found in the array." << endl;
    else
        cout << x << " is found in the array at index " << index << endl;
    system("pause");
    return 0;
}
```

Divide and Conquer Algorithms

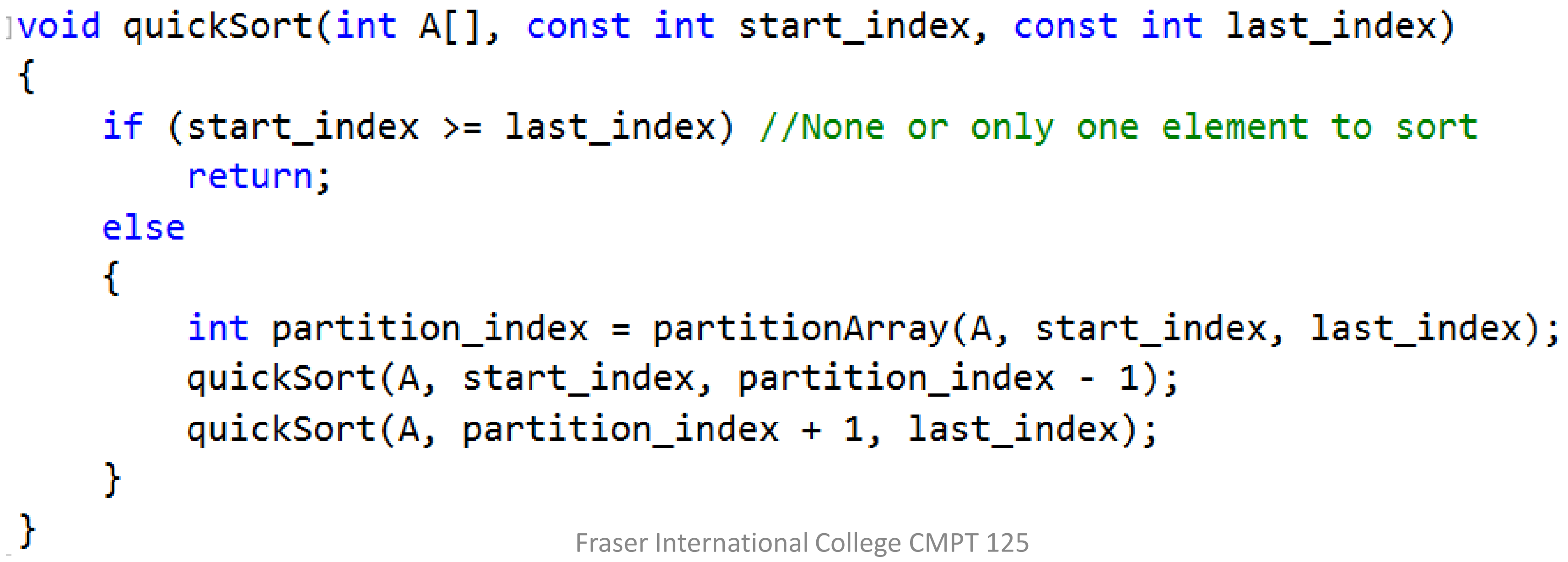
- Algorithms that are designed to solve a problem by first dividing the problem into several small parts, then solving each part independently, and finally combine the results of the parts to solve the original problem are known as divide and conquer algorithms
- Divide and conquer algorithms are usually implemented using recursively defined functions
- An example of divide and conquer algorithms is the **quick sort** algorithm described below

Quick Sort Algorithm

- Given an array/vector of some data type, the quick sort algorithm sorts the elements of the array as follows
  - **If the array/vector has only one or less elements then**
    - **Do nothing (it is already sorted)**
  - **Else**
    - **Partition the array/vector into two parts such that all the elements on the left side are less than the elements on the right side**
    - **Sort each part separately (using the same algorithm)**

Quick Sort Algorithm

- Thus the following function definition implements the quick sort algorithm as a recursive function
- Of course we need to define the **partitionArray** function that will partition the array/vector and return the index in the array/vector that partitions the array/vector into left and right side halves



```
void quickSort(int A[], const int start_index, const int last_index)
{
    if (start_index >= last_index) //None or only one element to sort
        return;
    else
    {
        int partition_index = partitionArray(A, start_index, last_index);
        quickSort(A, start_index, partition_index - 1);
        quickSort(A, partition_index + 1, last_index);
    }
}
```

Quick Sort Algorithm

- In order to partition the array/vector, all we need is to choose a certain value and then put all the elements of the array/vector that are less than the chosen value on the left side and the remaining elements of the array/vector on the right side
- The value we choose to partition the array/vector into two halves is known as the **pivot**
- We may chose any value to be a pivot so long as we think the chosen pivot will partition the array/vector into two roughly equal sized parts
- The ideal choice for a pivot is therefore the median of the elements of the array/vector
- However computing the median adds more complexity to the algorithm and therefore in practice any element of the array/vector is usually chosen to be a pivot
- In our discussion, we will use the last element of the array as a pivot

Quick Sort Algorithm

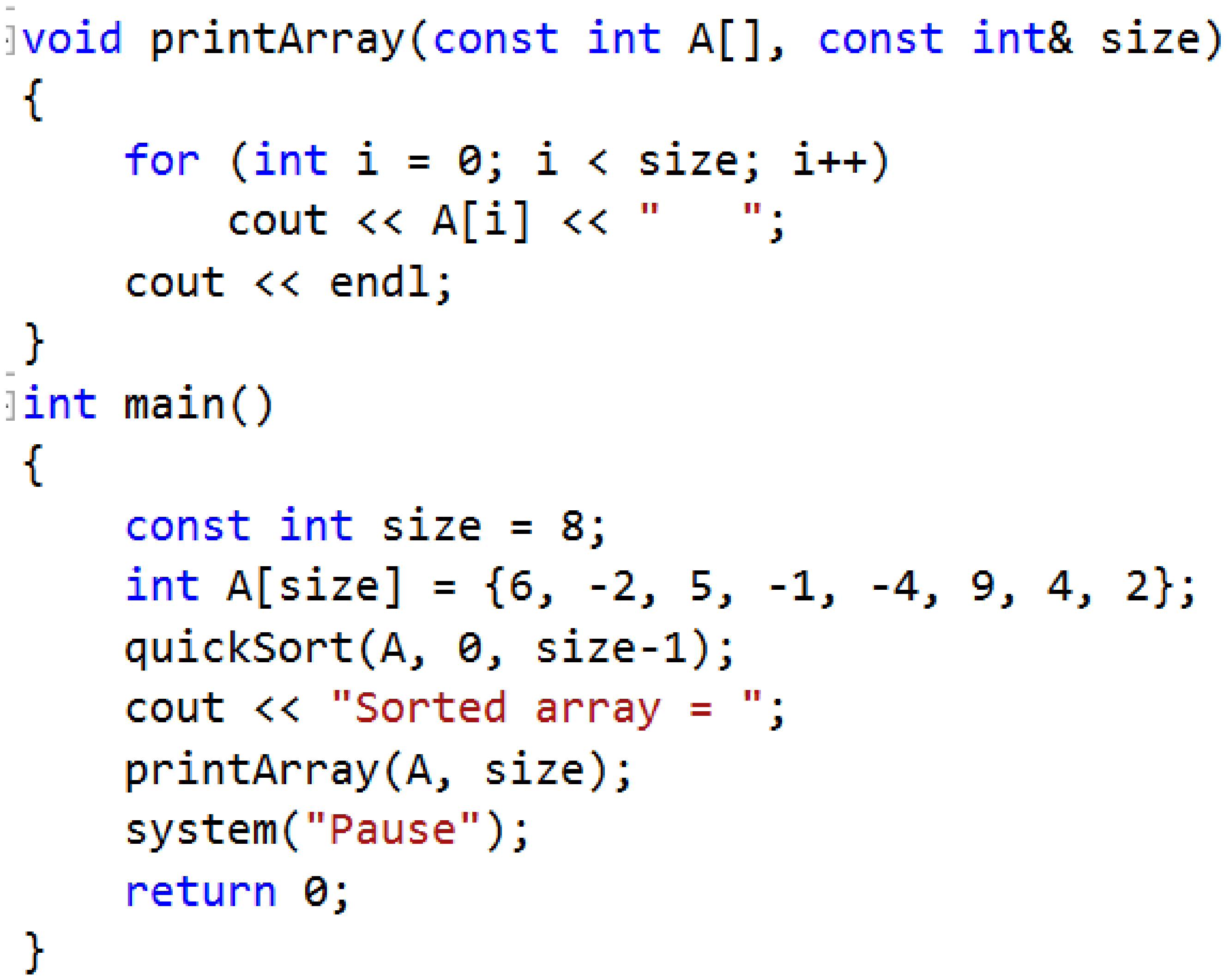
- The following function therefore will partition a given array into two parts

```
void swap(int &x, int &y)
{
     int temp = x;
     x = y;
     y = temp;
}
int partitionArray(int A[], const int start_index, const int last_index)
{
    int pivot = A[last_index];
    int partition_index = start_index - 1;
    for (int i = start_index; i < last_index; i++)
    {
        if (A[i] < pivot)
        {
            partition_index++;
            swap(A[partition_index], A[i]);
        }
    }
    partition_index++;
    swap(A[partition_index], A[last_index]);
    return partition_index;
}
```

Quick Sort Algorithm

- The partition function shown above partitions the array such that all the elements less than the pivot will be on the left side of the pivot and all the elements of the array greater than or equal to the pivot will be on the right side of the pivot
- Most importantly, the pivot will be in its correct position after the partition
- Thus all we need to do after partition is to sort each partition separately excluding the pivot as shown below
- This is why the **partitionArray** function is designed to return the index of the pivot element after the partition process so that we will know which elements are on the left side of the pivot and which ones on the right
- The following program tests the quick sort algorithm…

Quick Sort Algorithm



```
void printArray(const int A[], const int& size)
{
    for (int i = 0; i < size; i++)
        cout << A[i] << "   ";
    cout << endl;
}
int main()
{
    const int size = 8;
    int A[size] = {6, -2, 5, -1, -4, 9, 4, 2};
    quickSort(A, 0, size-1);
    cout << "Sorted array = ";
    printArray(A, size);
    system("Pause");
    return 0;
}
```

Complexity of the Quick Sort Algorithm

- The critical operation of the quick sort algorithm is the ***if (A[i] < pivot)*** operation
- Define ***f(n)*** to be a mathematical function designed to count how many times the critical operation is executed when sorting an array of size **n**
- Then we can easily see that

f(n) = (n-1) + f(partition1 size) + f(partition2 size)

- This is because the critical operation will be executed ***n-1*** times during partition and then ***f(partition size)*** when we sort each partition

Complexity of the Quick Sort Algorithm

- Although the partition sizes will depend on the actual array being sorted and the pivot, on average we may assume that each partition will have roughly ***n/2*** elements
- In this case

$$f(n) = (n-1) + 2f(n/2)$$

- This gives ***O(n log(n))*** complexity for the quick sort algorithm on average
- Of course in the worst case scenario, the pivot might happen to be the smallest or the largest element of the array
- In such cases, the partition will give **n-1** elements on one side and no elements on the other side
- Under such extreme cases, $f(n) = (n-1) + f(n-1)$ and the complexity will be $O(n^2)$