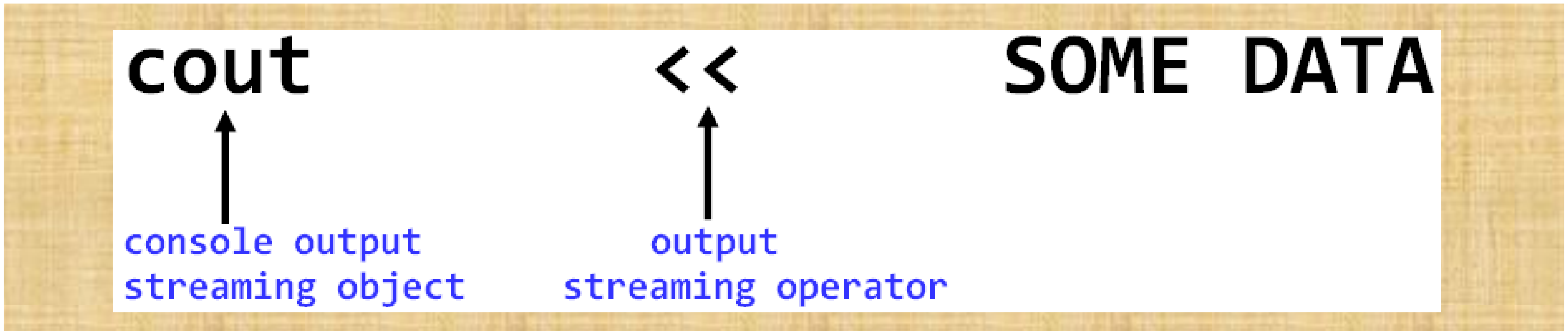
File Input Output
Exception Handling
and switch-case statements

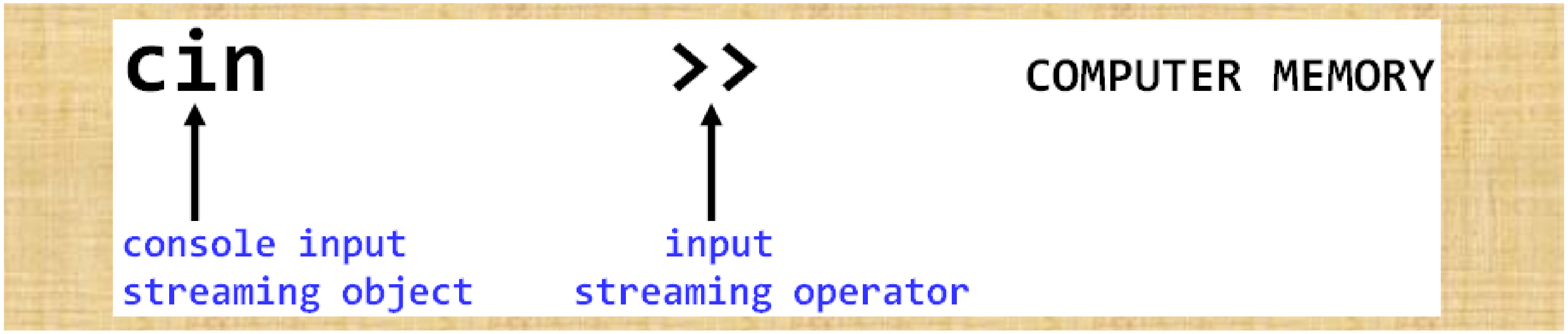
In this week

- C++ input/output streams
- C++ input/output streaming objects
- C++ input/output streaming operators
- Writing to output files
- Reading from input files
- Exception Handling in C++

C++ Input/Output Streams

- So far we have been working with console input (keyboard) and console output (screen) streams
- These were achieved with the **#include <iostream>** directive





C++ Input/Output Streams

- In addition to console input/output streams, C++ supports file input/output streams
- These are achieved with the **#include <fstream>** directive
- In order to work with file input/output streams, we need C++ file streaming objects that get connected to physical files stored in the hard drive of a computer
- Once file input/output streaming objects are defined, they are used together with the input and output streaming operators (**>>** and **<<** respectively) in the same way as the **cin** and **cout** console streaming objects

C++ Output File Streaming Objects

- In order to create an output file streaming object, we construct an object of type **ofstream**
- **ofstream** stands for output file stream
- **Syntax**

```
ofstream fout;
```

- This constructs a **default** output file streaming object
- In order to connect it to a file, we use the **open** member function as follows
- **Windows**

```
fout.open("C:/CMPT135/Week6/MyOutputFile.txt");
```

- **Mac OS**

```
fout.open("/Users/username/CMPT135/Week6/MyOutputFile.txt");
```

C++ Output File Streaming Objects

- Construction of a file output streaming object and connecting it to a file may also be performed using a non-default constructor as follows

```
ofstream fout("C:/CMPT135/Week6/MyOutputFile.txt");
```

- When we connect an output file streaming object to a file in the hard drive, then a **new empty file will be created automatically**
- **If a file with the same file name already exists in the same folder, then it will be overwritten (i.e. the existing file will be deleted and a new empty file with the same file name will be created)**
- When connecting a file streaming object to a file, if we specify only a file name without specifying the full path of the file, then most compilers will by default use the full path of our program source code file for the streaming object

C++ Output File Streaming Objects

- After we finish using an output file streaming object, we need to close it; that is, disconnect it from the physical file in the hard drive
- In order to do so, we use the **close** member function as follows

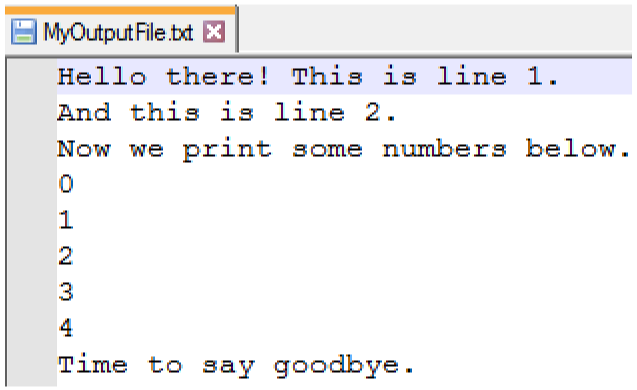
```
fout.close();
```

- Once an output file streaming object is closed, it can no more be used to write data to a file because it is disconnected from the file
- In order to re-use the file streaming object again to write data to a file, we need to re-connect it to the same or different file using the **open** member function

C++ Output File Streaming Objects

- The following program demonstrates, the construction, usage and closing of output file streaming object
- The created text file is also shown.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ofstream fout("C:/CMPT135/Week6/MyOutputFile.txt");
    fout << "Hello there! This is line 1." << endl;
    fout << "And this is line 2." << endl;
    fout << "Now we print some numbers below." << endl;
    for (int k = 0; k < 5; k++)
        fout << k << endl;
    fout << "Time to say goodbye." << endl;
    fout.close();
    system("Pause");
    return 0;
}
```



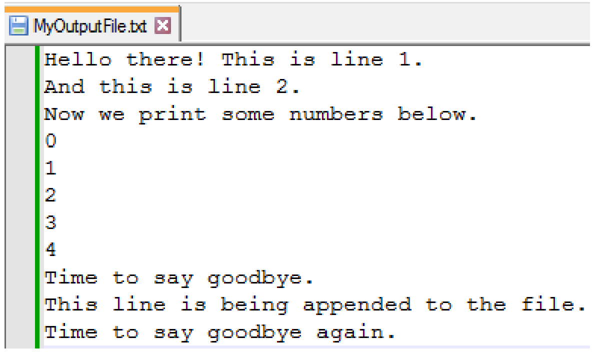
```
Hello there! This is line 1.
And this is line 2.
Now we print some numbers below.
0
1
2
3
4
Time to say goodbye.
```

C++ Output File Streaming Objects

- When we connect an output file streaming object to a file, if a file with the same file name exists in the same folder, then it will be overwritten
- We may however sometimes need to open an existing file but we don’t want to overwrite it
- Instead we may want to append data to it
- This is done by informing the output streaming object our intention to append data when we connect it to a file

C++ Output File Streaming Objects

- In order to append new information to an existing file, we do
  ofstream fout("C:/CMPT135/Week6/MyOutputFile.txt", ios::app);
- Now fout will append data to the contents of the file
- The following program appends some data to an existing file. Assume the file exists in the same folder where the cpp file is located. The existing file is appended as expected as shown below.



```
Hello there! This is line 1.
And this is line 2.
Now we print some numbers below.
0
1
2
3
4
Time to say goodbye.
This line is being appended to the file.
Time to say goodbye again.
```

```cpp
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ofstream fout("C:/CMPT135/Week6/MyOutputFile.txt", ios::app);
    fout << "This line is being appended to the file." << endl;
    fout << "Time to say goodbye again." << endl;
    fout.close();
    system("Pause");
    return 0;
}
```

C++ Input File Streaming Objects

- We can also construct a file input stream object and read data from a file to our program
- **Syntax**

```
ifstream fin;
```

- This constructs a **default** file input streaming object that is not connected to any file
- In order to connect it to a file, we use the **open** member function as follows

```
fin.open("C:/CMPT135/Wee6/MyInputFile.txt");
```

- We may also construct a file input streaming object and connect it to a file at the same time using a non-default constructor as follows

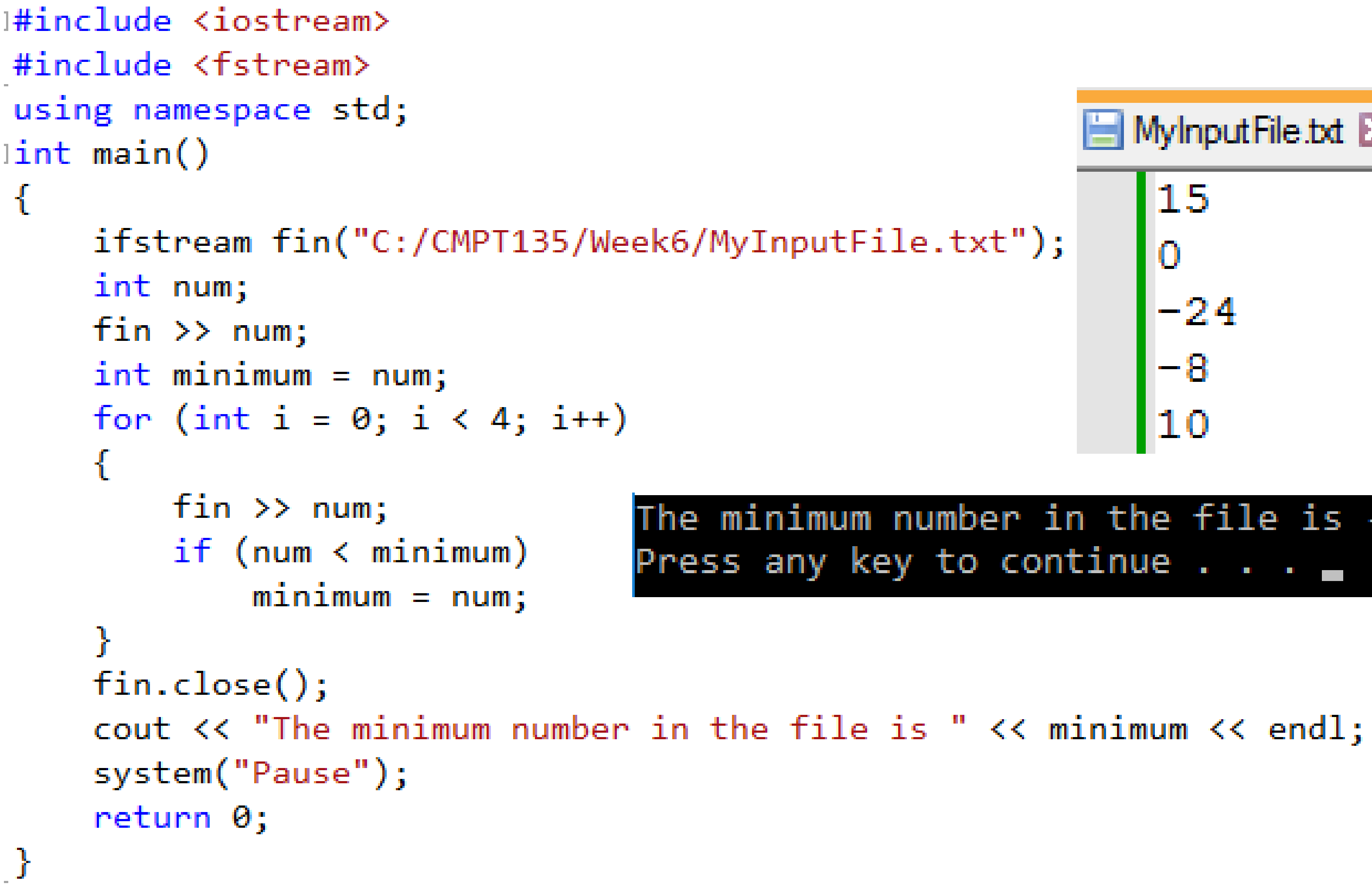
```
ifstream fin("C:/CMPT135/Wee6/MyInputFile.txt");
```

- Of course we have to first make certain that such a file exists

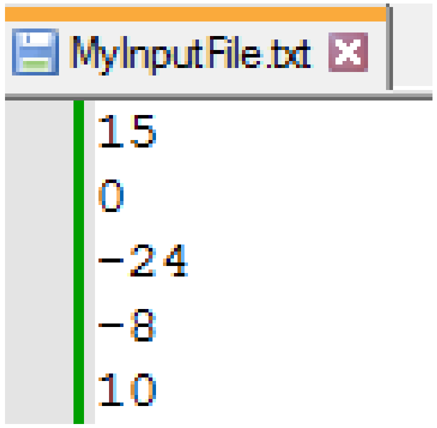
C++ Input File Streaming Objects

- For example, suppose that there is a file named **C:/CMPT135/Week6/MyInputFile.txt** stored in the hard drive of the computer
- Moreover suppose the file contains five integer numbers separated either by spaces, tabs or each number on its own line
- Then the following program will read the five numbers in the file one by one using a loop and then compute the minimum number in the file and print it to the screen
- A sample input file and its corresponding output are also shown to demonstrate the program

C++ Input File Streaming Objects

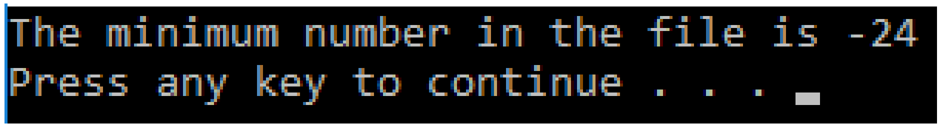


```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream fin("C:/CMPT135/Week6/MyInputFile.txt");
    int num;
    fin >> num;
    int minimum = num;
    for (int i = 0; i < 4; i++)
    {
        fin >> num;
        if (num < minimum)
            minimum = num;
    }
    fin.close();
    cout << "The minimum number in the file is " << minimum << endl;
    system("Pause");
    return 0;
}
```



MyInputFile.txt

```
15
0
-24
-8
10
```



```
The minimum number in the file is -24
Press any key to continue . . .
```

C++ Input File Streaming Objects

- Extra care needs to be followed when working with file input stream objects
  - Firstly, we should be certain the physical file from which to read exists
  - Secondly, we should know the type of the data in the file in order to read to proper data type variables
  - Thirdly, we should ascertain there is enough data in the file to read

Checking File IO Runtime Errors

- It is important we check if an input/output file stream object is opened correctly before using it
- Sometimes, output stream object may fail because we don't have the privilege to create a new file on the computer we are using
- An input file stream may fail because there is no file with the given file name in the folder we are specifying
- In order to check the success of input/output file stream objects we proceed as follows:

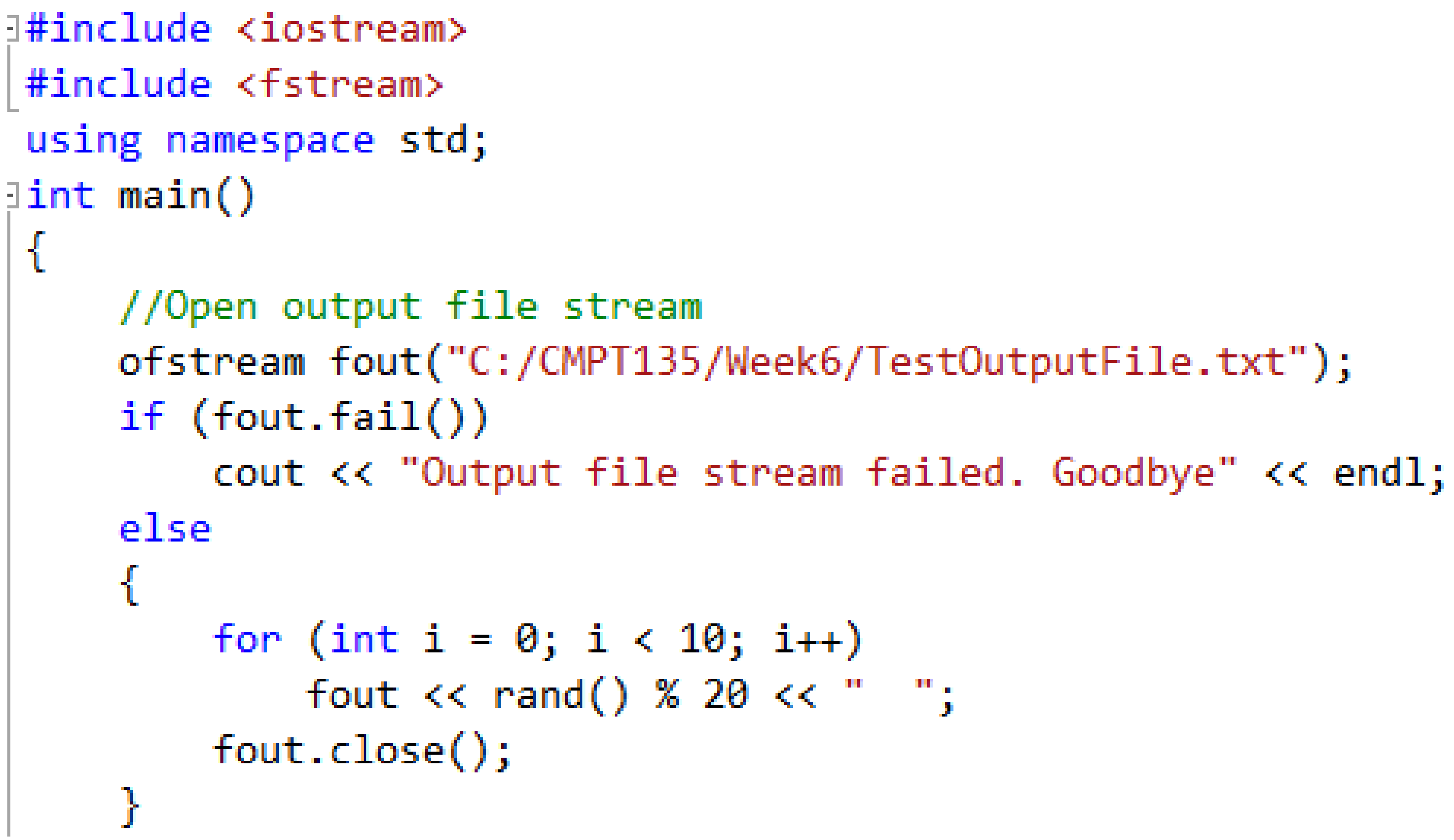
```
ofstream fout("Test.txt")
if (fout.fail())
    //File opening failed
else
    //File opening succeded
```

```
ifstream fin("Test.txt")
if (fin.fail())
    //File opening failed
else
    //File opening succeded
```

Checking File IO Runtime Errors

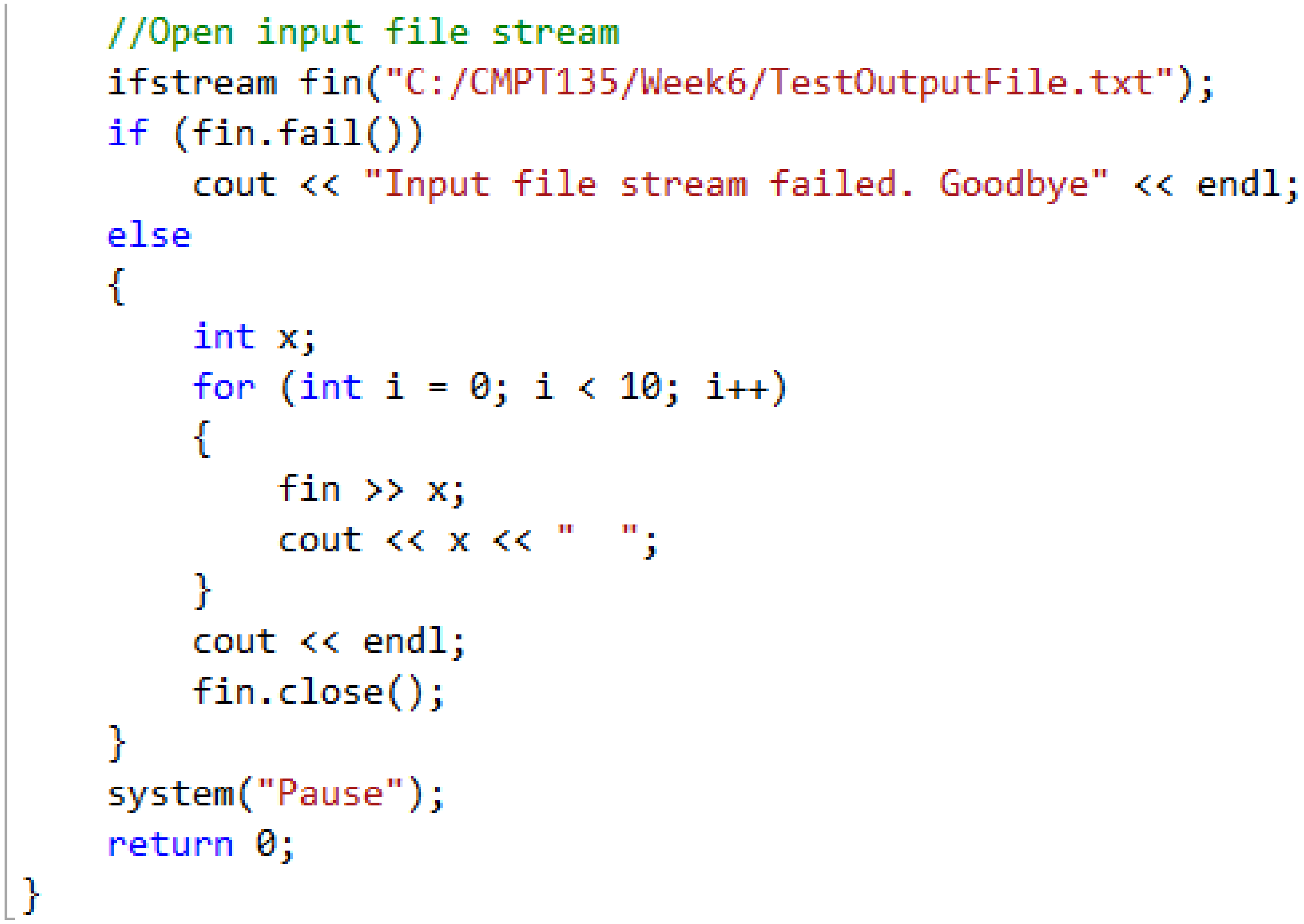
- In the following example, we give a complete example to create a new file, write some integer numbers in the file, and then read the numbers from the file and print the numbers on to the screen
- We also check the success of our file input and file output streaming objects before we make use of the objects

Checking File IO Runtime Errors



```cpp
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    //Open output file stream
    ofstream fout("C:/CMPT135/Week6/TestOutputFile.txt");
    if (fout.fail())
        cout << "Output file stream failed. Goodbye" << endl;
    else
    {
        for (int i = 0; i < 10; i++)
            fout << rand() % 20 << "  ";
        fout.close();
    }
```

Checking File IO Runtime Errors



```
    //Open input file stream
    ifstream fin("C:/CMPT135/Week6/TestOutputFile.txt");
    if (fin.fail())
        cout << "Input file stream failed. Goodbye" << endl;
    else
    {
        int x;
        for (int i = 0; i < 10; i++)
        {
            fin >> x;
            cout << x << "  ";
        }
        cout << endl;
        fin.close();
    }
    system("Pause");
    return 0;
}
```

More General Complete Example

- Create a text file named **ClassList.txt** manually using notepad and edit it as follows and save it somewhere on the computer

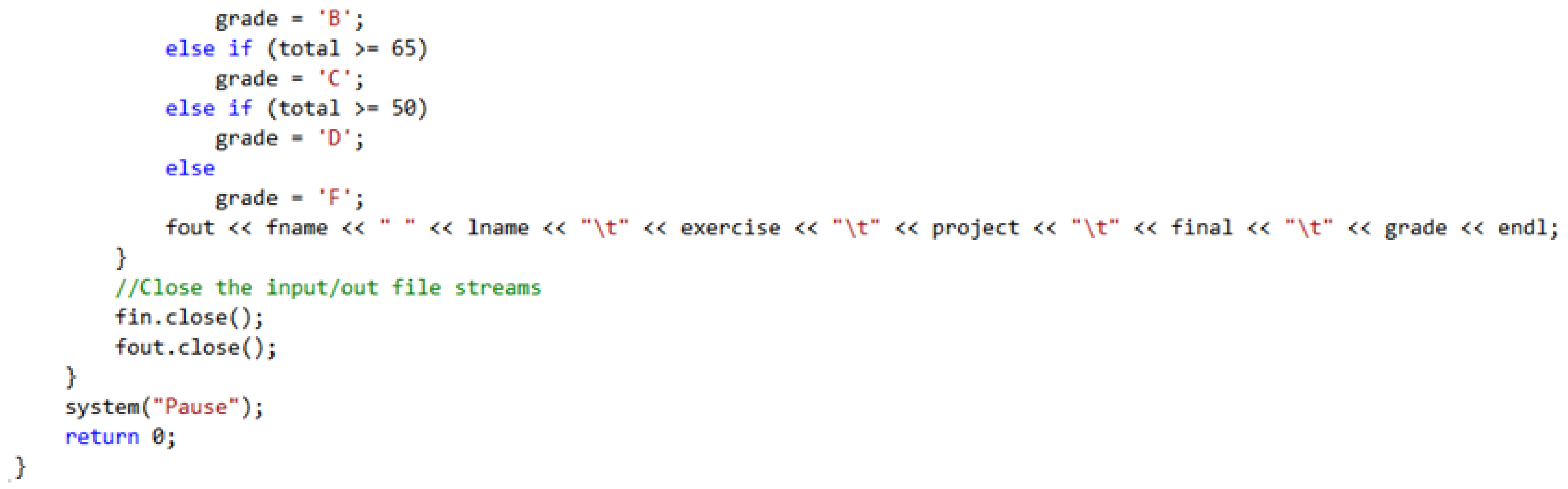
John Walter	**20**	**19**	**45**
Sara Gill	**16**	**15**	**35**
Mark Black	**23**	**24**	**50**
Jess Paul	**10**	**20**	**25**
Joe Nash	**14**	**18**	**44**

Think of these as students' full names and their assessment marks for exercises, projects and final exam. Write a C++ program that reads this file (ClassList.txt) and that creates a new file named **Report.txt** on the same folder with the same content as the input file together with the letter grades of the students on the same line for each student. For the letter grades, use the settings [90, 100] = A, [75, 90) = B, [65, 75) = C, [50, 65) = D and [0, 50) = F.

More General Complete Example

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
int main()
{
    ifstream fin("C:/CMPT135/Week6/ClassList.txt");
    ofstream fout("C:/CMPT135/Week6/Report.txt");
    if (fin.fail())
        cout << "Input file not found. Goodbye." << endl;
    else if (fout.fail())
        cout << "Output file not found. Goodbye." << endl;
    else
    {
        //Now we have both the input and output files ready to use
        string fname, lname;
        int exercise, project, final, total;
        char grade;
        for (int i = 0; i < 5; i++)
        {
            fin >> fname >> lname >> exercise >> project >> final;
            total = exercise + project + final;
            if (total >= 90)
                grade = 'A';
            else if (total >= 75)
```

More General Complete Example



```
                grade = 'B';
            else if (total >= 65)
                grade = 'C';
            else if (total >= 50)
                grade = 'D';
            else
                grade = 'F';
            fout << fname << " " << lname << "\t" << exercise << "\t" << project << "\t" << final << "\t" << grade << endl;
        }
        //Close the input/out file streams
        fin.close();
        fout.close();
    }
    system("Pause");
    return 0;
}
```

- Observe that the string data in the input file should be read into string data variables while the integer data should be read into int data variables
- Moreover, we don't need to read all the data in the input file and store them in some containers such as arrays because we can process one student data at a time
- That is we can simply read one student information and then print the same information together with the letter grade of the student on to the output file and then proceed to the next student

Input File End of File Marker

- Sometimes we may wish to read input data from an input file but we may not know the amount of data that is stored in the file
- In such cases, we need to read from the file until all the data in the file is read
- That is, until we reach the END OF THE FILE
- File input streaming objects can test the end of a file using **eof()** member function that returns true when the end of file is reached

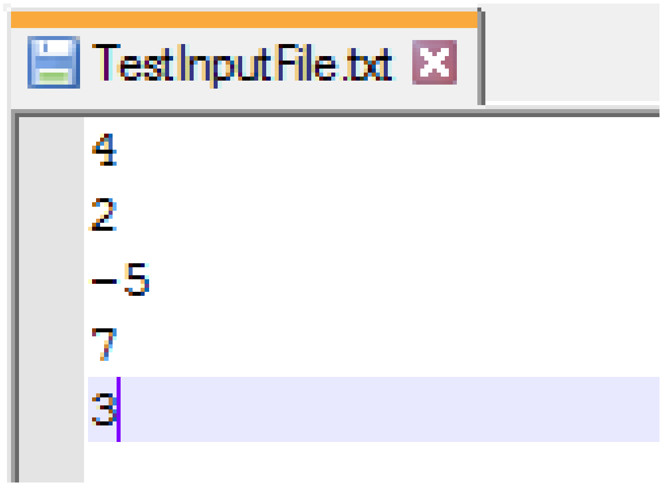
Input File End of File Marker

- When using the **eof()** member function of a file input streaming object, it is important not to have any empty lines at the end of the input file; otherwise the empty lines will be read before we can reach the end of file marker
- In order to demonstrate the **eof** member function, assume that we have an input file named **TestInputFile.txt**
- Assume also that this input file contains some integer numbers with no any empty lines at the end of the file (but that we don't know how many numbers are in the file)
- Then the following program will read all the numbers in the file and print the minimum and maximum values in the file

Input File End of File Marker

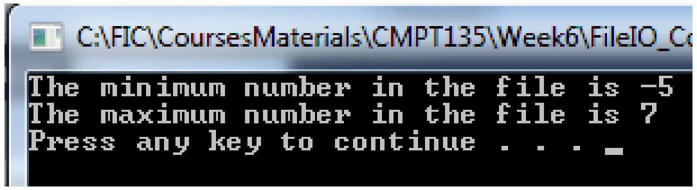
```
int main()
{
    ifstream fin("C:/CMPT135/Week6/TestInputFile.txt");
    if (fin.fail())
    {
        cout << "Input file not found. Exiting program." << endl;

    }
    else
    {
        //Read the first number input
        int a;
        fin >> a;

        //At this point, this first number is both the min and the max
        int minimum = a, maximum = a;

        //Now read all the remaining numbers using eof marker
        while (fin.eof() == false)
        {
            fin >> a;
            if (a > maximum)
                maximum = a;
            if (a < minimum)
                minimum = a;
        }
        fin.close();

        //Print the minimum and maximum
        cout << "The minimum number in the file is " << minimum << endl;
        cout << "The maximum number in the file is " << maximum << endl;
    }
    system("Pause");
    return 0;
}
```

Input File Example


TestInputFile.txt

```
4
2
-5
7
3
```

Output of the program


C:\FIC\CoursesMaterials\CMPT135\Week6\FileIO_C

```
The minimum number in the file is -5
The maximum number in the file is 7
Press any key to continue . . .
```

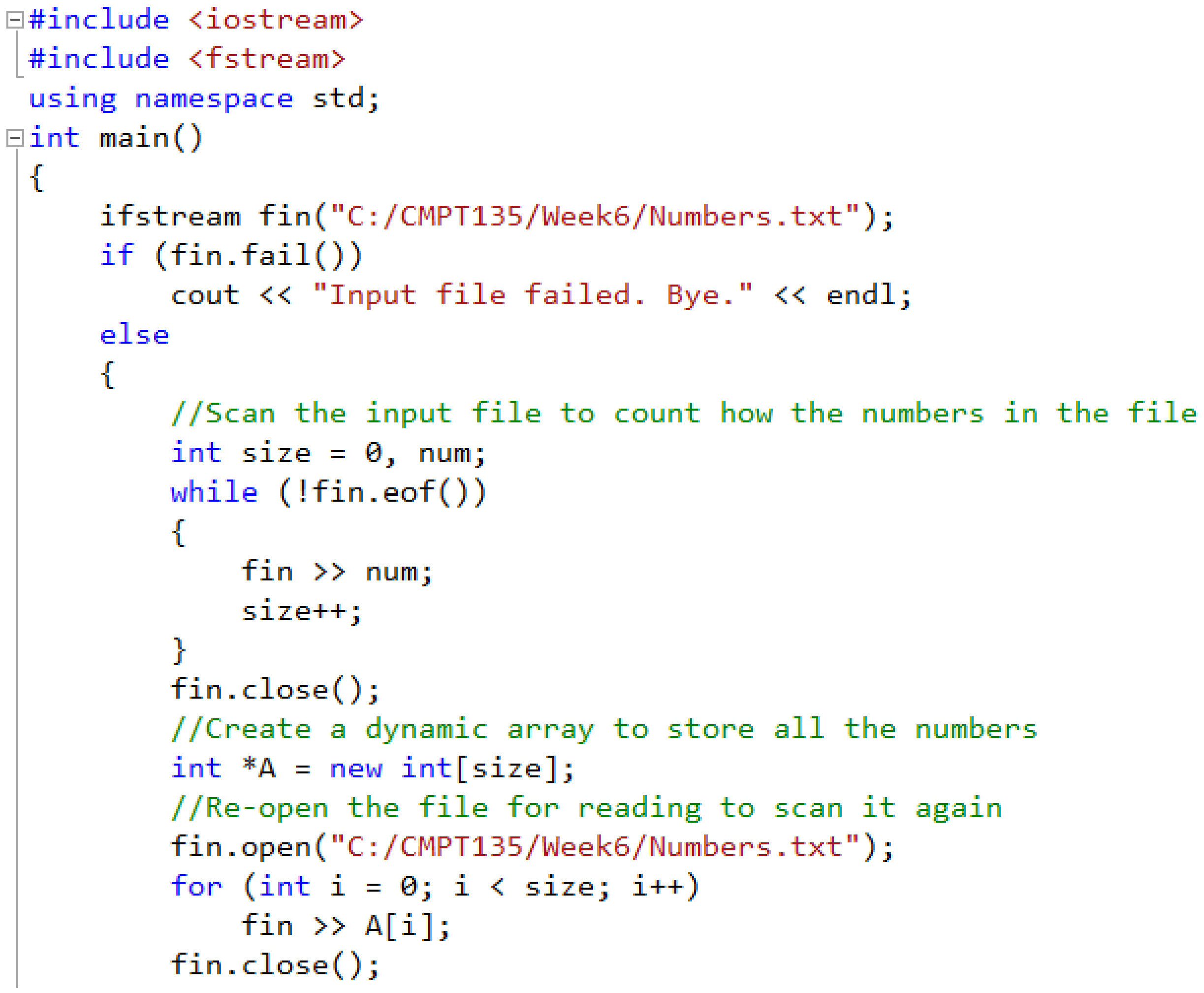
Input File End of File Marker

- **Example:-** Given an input file named **Numbers.txt** that contains several integer numbers, write a C++ program that edits (i.e. modifies) the file so that the numbers in the file are reversed. Assume there are no any empty lines at the end of the file.
- **Hint:-** We need to read from a file and write on to the same file
- We can solve this problem in any one of the following three ways
  - ➢ **Algorithm 1. Using a dynamic array**
  - ➢ **Algorithm 2. Using the SmartArray class**
  - ➢ **Algorithm 3. Using a recursive function**

Input File End of File Marker

Algorithm 1

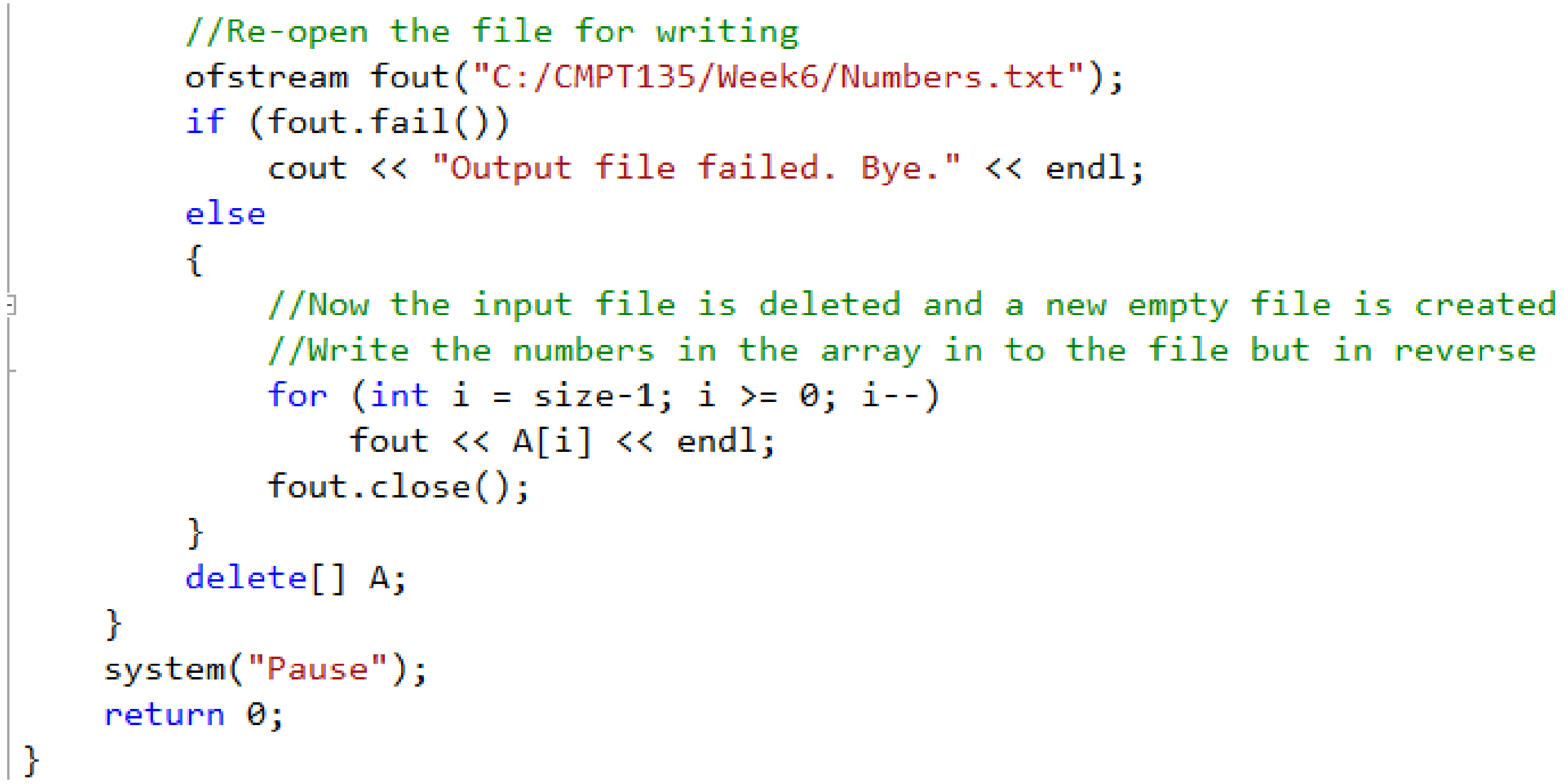
Using a dynamic array



```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream fin("C:/CMPT135/Week6/Numbers.txt");
    if (fin.fail())
        cout << "Input file failed. Bye." << endl;
    else
    {
        //Scan the input file to count how the numbers in the file
        int size = 0, num;
        while (!fin.eof())
        {
            fin >> num;
            size++;
        }
        fin.close();
        //Create a dynamic array to store all the numbers
        int *A = new int[size];
        //Re-open the file for reading to scan it again
        fin.open("C:/CMPT135/Week6/Numbers.txt");
        for (int i = 0; i < size; i++)
            fin >> A[i];
        fin.close();
```

Input File End of File Marker

- Continuation…

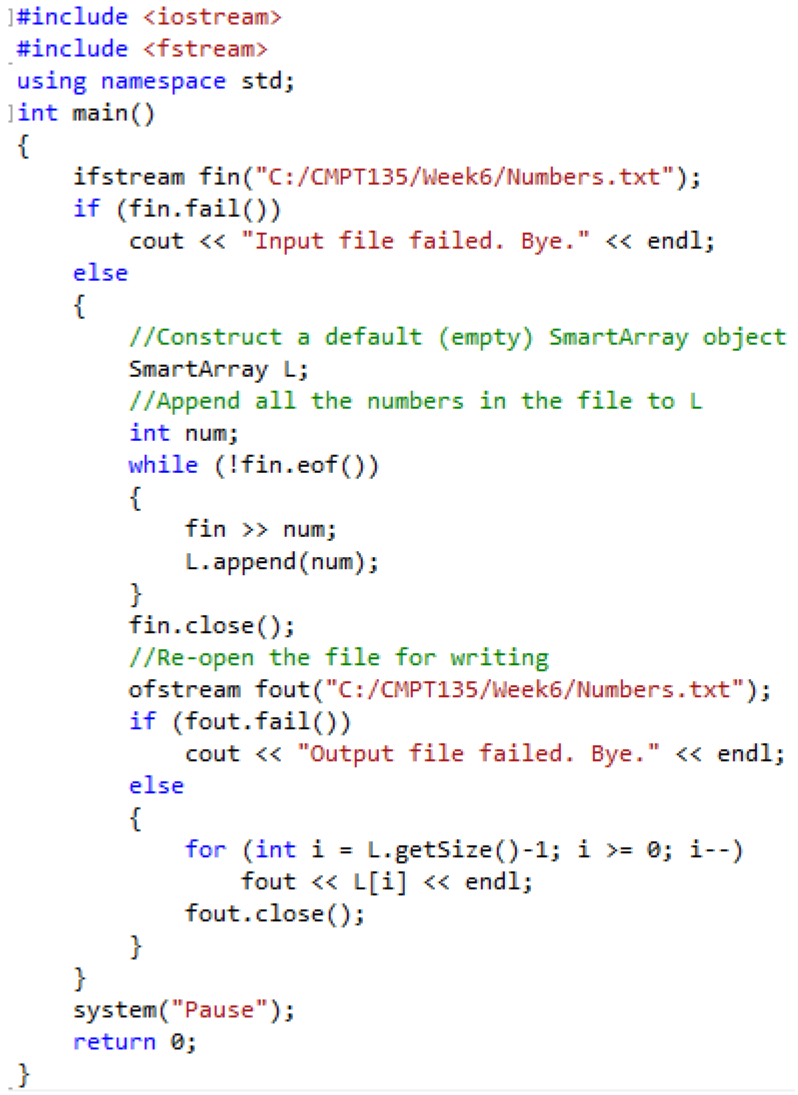


```
        //Re-open the file for writing
        ofstream fout("C:/CMPT135/Week6/Numbers.txt");
        if (fout.fail())
            cout << "Output file failed. Bye." << endl;
        else
        {
            //Now the input file is deleted and a new empty file is created
            //Write the numbers in the array in to the file but in reverse
            for (int i = size-1; i >= 0; i--)
                fout << A[i] << endl;
            fout.close();
        }
        delete[] A;
    }
    system("Pause");
    return 0;
}
```

Input File End of File Marker

<u>Algorithm 2</u>

Using the SmartArray class

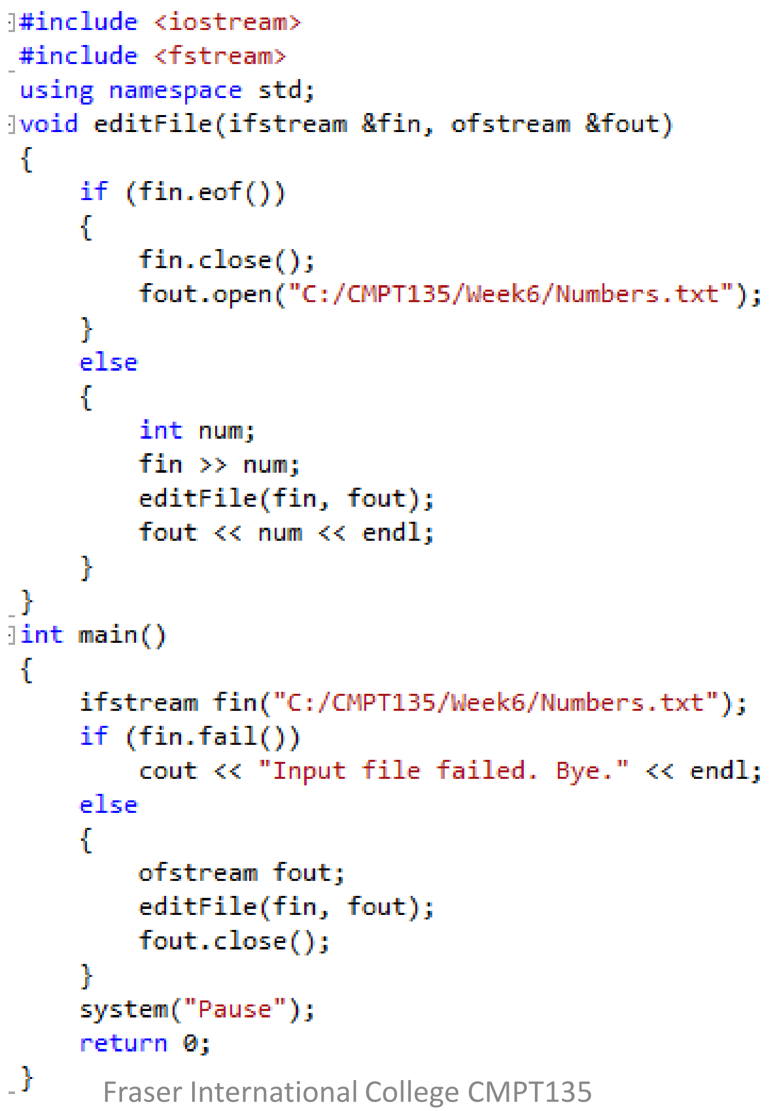


```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream fin("C:/CMPT135/Week6/Numbers.txt");
    if (fin.fail())
        cout << "Input file failed. Bye." << endl;
    else
    {
        //Construct a default (empty) SmartArray object
        SmartArray L;
        //Append all the numbers in the file to L
        int num;
        while (!fin.eof())
        {
            fin >> num;
            L.append(num);
        }
        fin.close();
        //Re-open the file for writing
        ofstream fout("C:/CMPT135/Week6/Numbers.txt");
        if (fout.fail())
            cout << "Output file failed. Bye." << endl;
        else
        {
            for (int i = L.getSize()-1; i >= 0; i--)
                fout << L[i] << endl;
            fout.close();
        }
    }
    system("Pause");
    return 0;
}
```

Input File End of File Marker

Algorithm 3

Using a recursive function



```cpp
#include <iostream>
#include <fstream>
using namespace std;
void editFile(ifstream &fin, ofstream &fout)
{
    if (fin.eof())
    {
        fin.close();
        fout.open("C:/CMPT135/Week6/Numbers.txt");
    }
    else
    {
        int num;
        fin >> num;
        editFile(fin, fout);
        fout << num << endl;
    }
}
int main()
{
    ifstream fin("C:/CMPT135/Week6/Numbers.txt");
    if (fin.fail())
        cout << "Input file failed. Bye." << endl;
    else
    {
        ofstream fout;
        editFile(fin, fout);
        fout.close();
    }
    system("Pause");
    return 0;
}
```

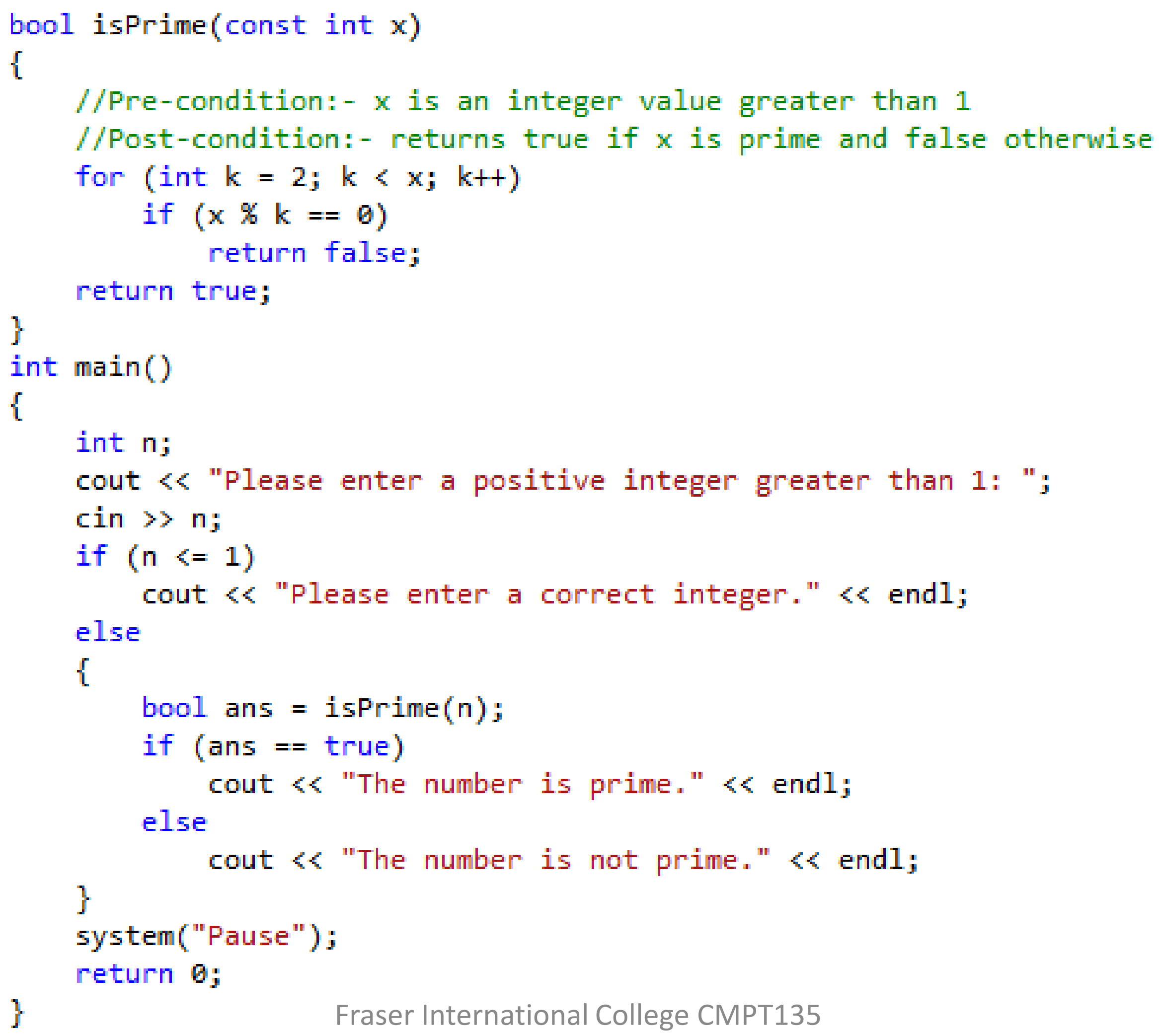
Exception Handling in C++

Motivation: Pre-condition and Post-condition

- Generally speaking it is assumed that the parameters of a function will always receive correct data values from their corresponding arguments
- This is why we need to check the correctness of any value before using it as argument when calling a function
- That is a C++ function assumes its parameter will not cause any runtime error or errors in the function body
- This is usually emphasized by explicitly writing comments inside the function block that specify what conditions the parameters of the function must satisfy (known as the pre-condition) and what the function is intended to perform (known as the post-condition)
- Whenever a function is given arguments that violate its pre-condition requirements, then the function may get into a runtime error and crash the program or may give some wrong result
- The next example shows ***isPrime*** function stating explicitly its pre-condition and post-condition for clarity purposes

Exception Handling in C++

Motivation: Pre-condition and Post-condition

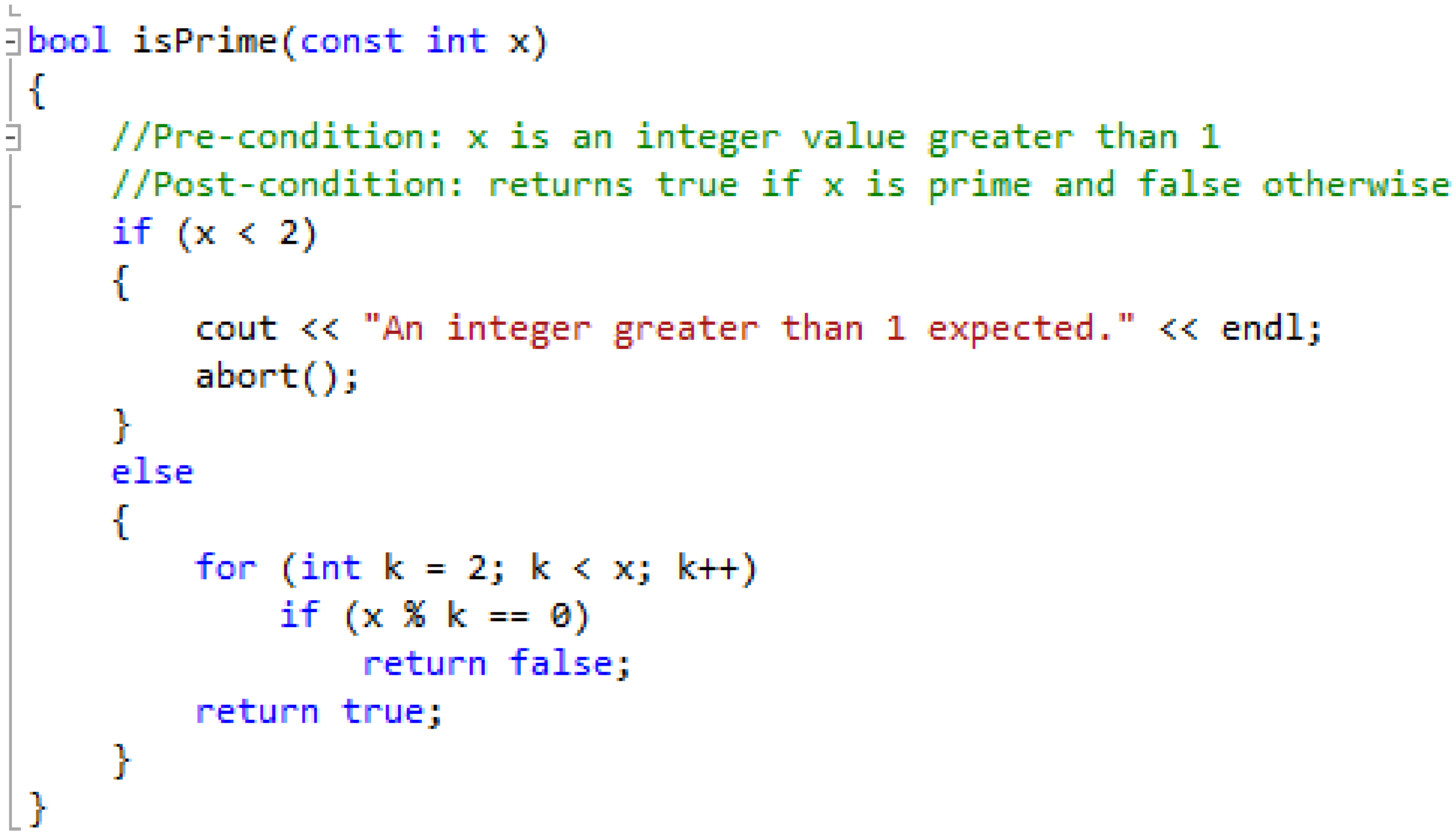


```
bool isPrime(const int x)
{
    //Pre-condition:- x is an integer value greater than 1
    //Post-condition:- returns true if x is prime and false otherwise
    for (int k = 2; k < x; k++)
        if (x % k == 0)
            return false;
    return true;
}
int main()
{
    int n;
    cout << "Please enter a positive integer greater than 1: ";
    cin >> n;
    if (n <= 1)
        cout << "Please enter a correct integer." << endl;
    else
    {
        bool ans = isPrime(n);
        if (ans == true)
            cout << "The number is prime." << endl;
        else
            cout << "The number is not prime." << endl;
    }
    system("Pause");
    return 0;
}
```

Exception Handling in C++

- But what if the parameters of a function violate the pre-condition?
- Obviously a runtime error may happen or may result to some semantic errors
- Sometimes, we may also opt to check the correctness of the parameters of a function and crash the program if the parameter doesn't satisfy the pre-condition
- One way to crash the program will be to use the **abort** function as shown below

Exception Handling in C++



```cpp
bool isPrime(const int x)
{
    //Pre-condition: x is an integer value greater than 1
    //Post-condition: returns true if x is prime and false otherwise
    if (x < 2)
    {
        cout << "An integer greater than 1 expected." << endl;
        abort();
    }
    else
    {
        for (int k = 2; k < x; k++)
            if (x % k == 0)
                return false;
        return true;
    }
}
```

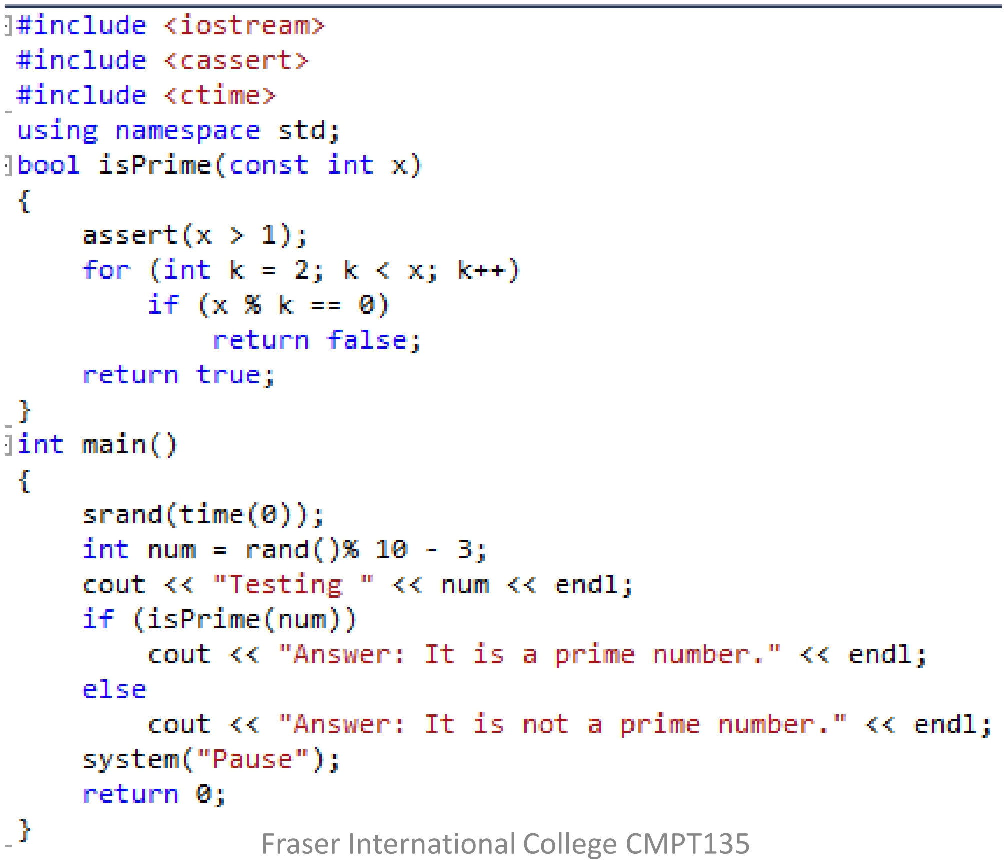
Exception Handling in C++

- Another way to test the parameter is to use the **assert** built-in function
- This function requires an include directive

  #include <cassert>
- The assert function takes one argument which is often a Boolean expression testing some condition
- If the Boolean expression is evaluated to true then the assert function does nothing and execution continues to the next statement
- If the Boolean expression is evaluated to false then the assert function prints a description of the error message and then aborts the program. See below

Exception Handling in C++

- The following program uses an assert built-in function to test a parameter value. Please note that we don't need to put an assert statement inside if-else blocks because the assert function will print its own error message whenever the assertion fails. Run the following program repeatedly and see what the outputs will be

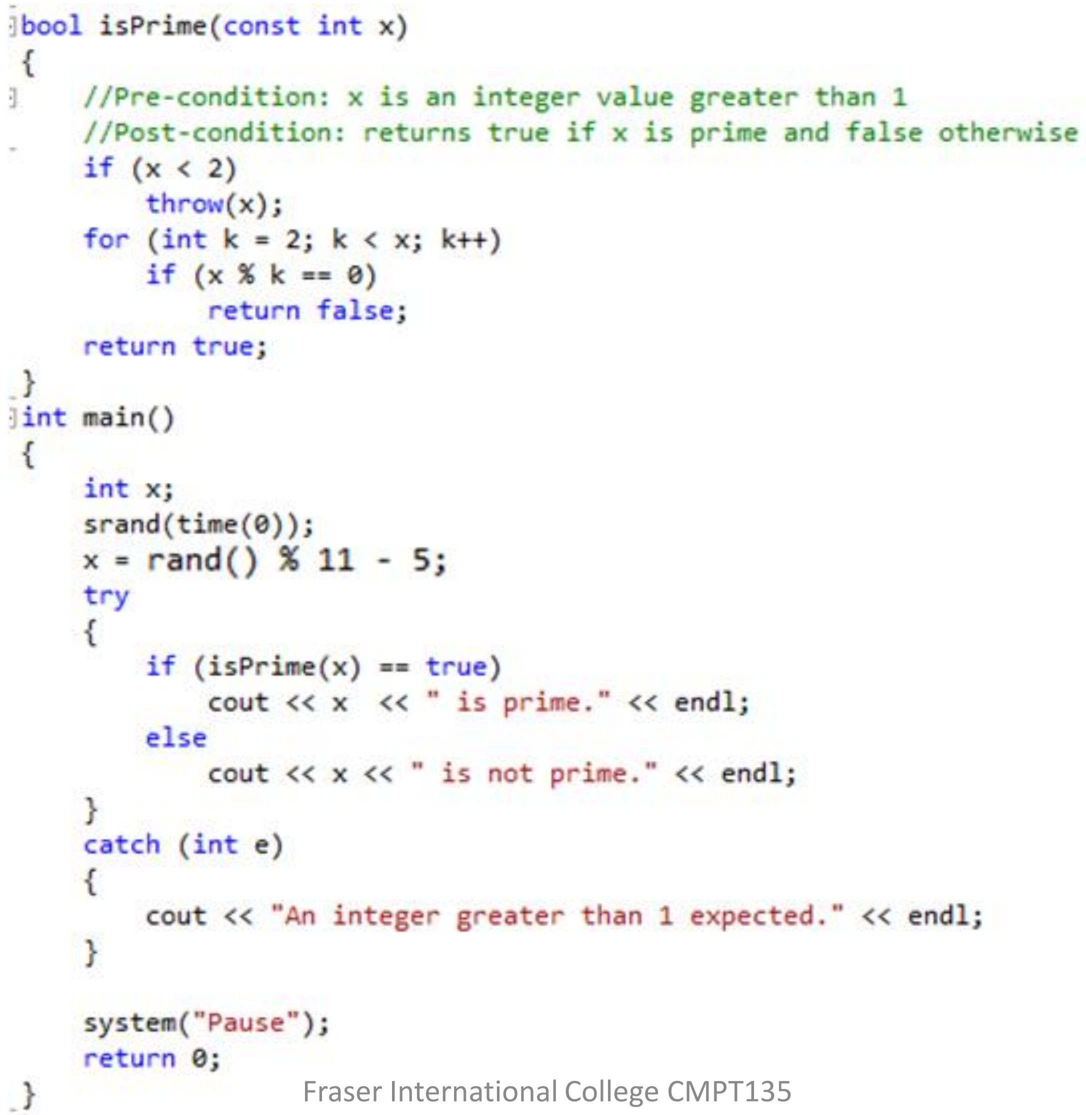


```
#include <iostream>
#include <cassert>
#include <ctime>
using namespace std;
bool isPrime(const int x)
{
    assert(x > 1);
    for (int k = 2; k < x; k++)
        if (x % k == 0)
            return false;
    return true;
}
int main()
{
    srand(time(0));
    int num = rand()% 10 - 3;
    cout << "Testing " << num << endl;
    if (isPrime(num))
        cout << "Answer: It is a prime number." << endl;
    else
        cout << "Answer: It is not a prime number." << endl;
    system("Pause");
    return 0;
}
```

Exception Handling in C++

- Yet another way to check the correctness of any variable (including a parameter) in a program is to use **try-catch blocks**
- In the try-catch block, the code that we need to execute is placed in a try block
- If any variable is not correct, we throw an exception and the code in the try block is automatically skipped and execution goes to the catch block which will handle the error
- See below

Exception Handling in C++



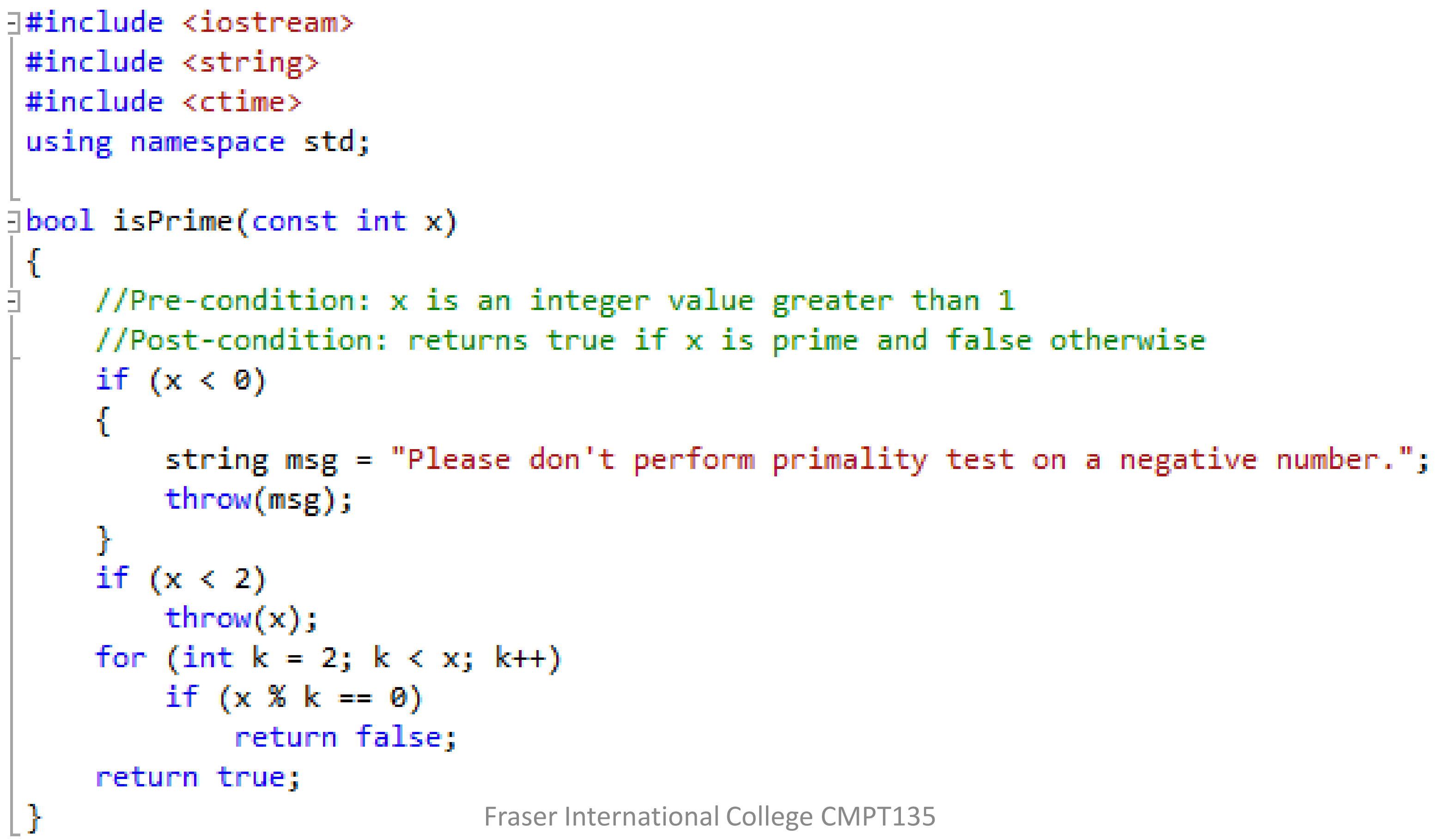
```
bool isPrime(const int x)
{
    //Pre-condition: x is an integer value greater than 1
    //Post-condition: returns true if x is prime and false otherwise
    if (x < 2)
        throw(x);
    for (int k = 2; k < x; k++)
        if (x % k == 0)
            return false;
    return true;
}
int main()
{
    int x;
    srand(time(0));
    x = rand() % 11 - 5;
    try
    {
        if (isPrime(x) == true)
            cout << x  << " is prime." << endl;
        else
            cout << x << " is not prime." << endl;
    }
    catch (int e)
    {
        cout << "An integer greater than 1 expected." << endl;
    }

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

Exception Handling in C++

- Please note that
  - The throw statement can throw any data type including struct type and class type objects
  - The catch block can catch any data type
  - If we are interested in different types of errors then we can throw different data types from the different parts of a program and then have multiple catch blocks such that each catch block catches different data types
  - This way every throw will be caught by a catch that corresponds the data type of the throw
- See the following example

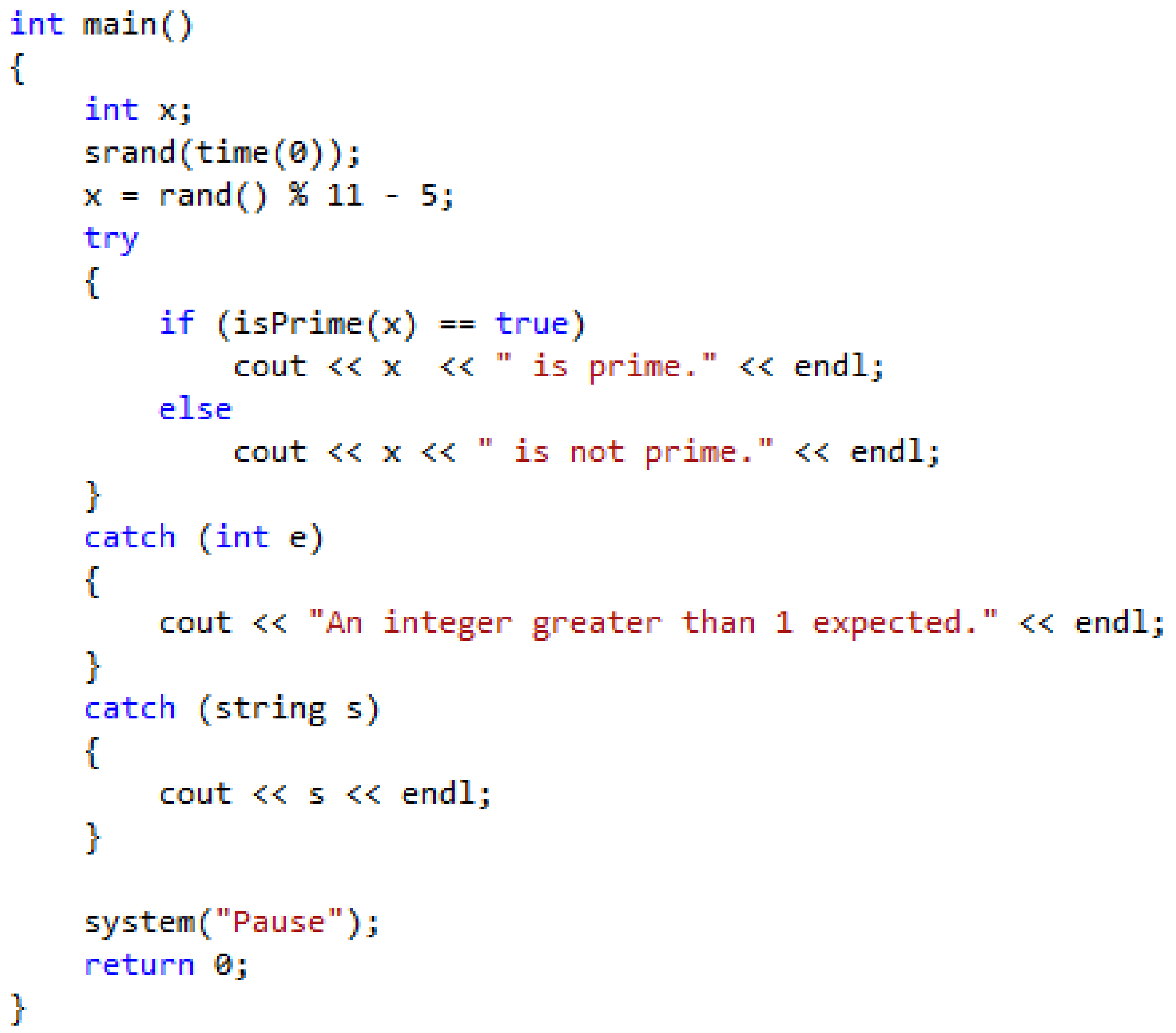
Exception Handling in C++

- See next slide for the main program



```cpp
#include <iostream>
#include <string>
#include <ctime>
using namespace std;

bool isPrime(const int x)
{
    //Pre-condition: x is an integer value greater than 1
    //Post-condition: returns true if x is prime and false otherwise
    if (x < 0)
    {
        string msg = "Please don't perform primality test on a negative number.";
        throw(msg);
    }
    if (x < 2)
        throw(x);
    for (int k = 2; k < x; k++)
        if (x % k == 0)
            return false;
    return true;
}
```

Exception Handling in C++



```cpp
int main()
{
    int x;
    srand(time(0));
    x = rand() % 11 - 5;
    try
    {
        if (isPrime(x) == true)
            cout << x  << " is prime." << endl;
        else
            cout << x << " is not prime." << endl;
    }
    catch (int e)
    {
        cout << "An integer greater than 1 expected." << endl;
    }
    catch (string s)
    {
        cout << s << endl;
    }

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

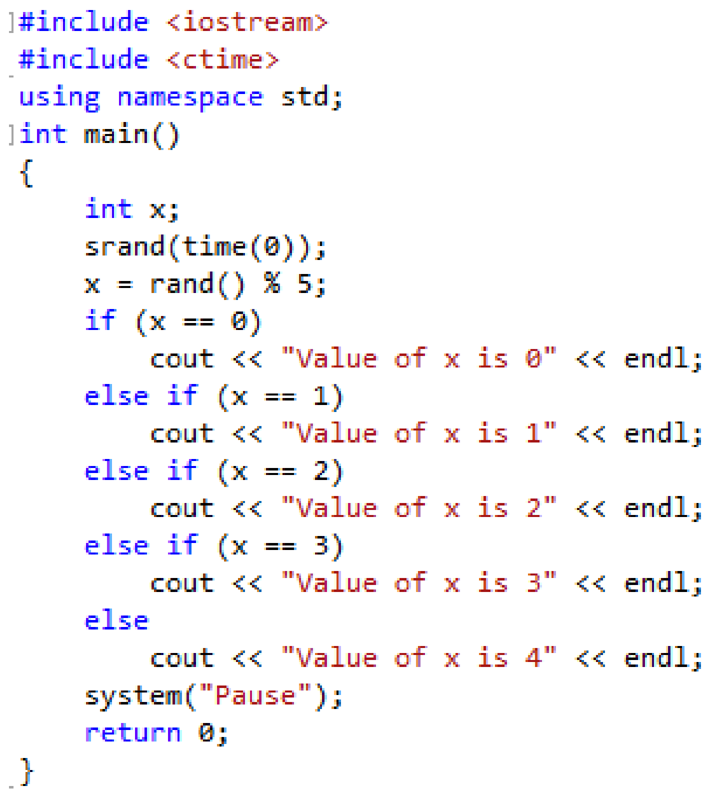
Exception Handling in C++

- Sometimes, we may require a catch block to catch any remaining data type that are not caught in the catch blocks listed
- This is similar to else block in conditional statements
- C++ allows such a catch whose syntax is given as follows

```
    catch(…)
```

C++ switch-case statements

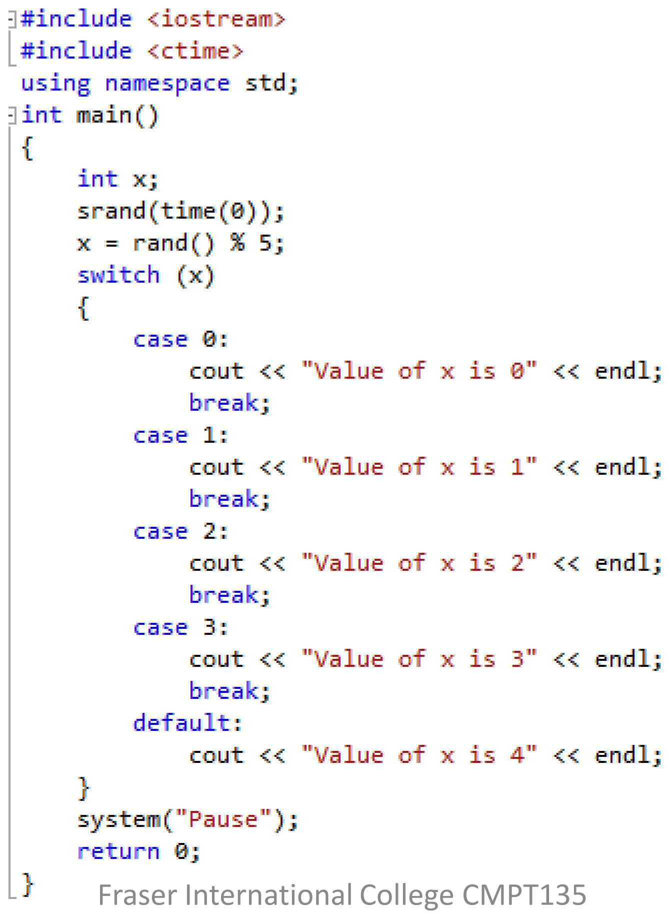
- Consider the following simple program with some conditional statements



```
#include <iostream>
#include <ctime>
using namespace std;
int main()
{
    int x;
    srand(time(0));
    x = rand() % 5;
    if (x == 0)
        cout << "Value of x is 0" << endl;
    else if (x == 1)
        cout << "Value of x is 1" << endl;
    else if (x == 2)
        cout << "Value of x is 2" << endl;
    else if (x == 3)
        cout << "Value of x is 3" << endl;
    else
        cout << "Value of x is 4" << endl;
    system("Pause");
    return 0;
}
```

C++ switch-case statements

- C++ provides the **switch-case** statement to achieve the same result



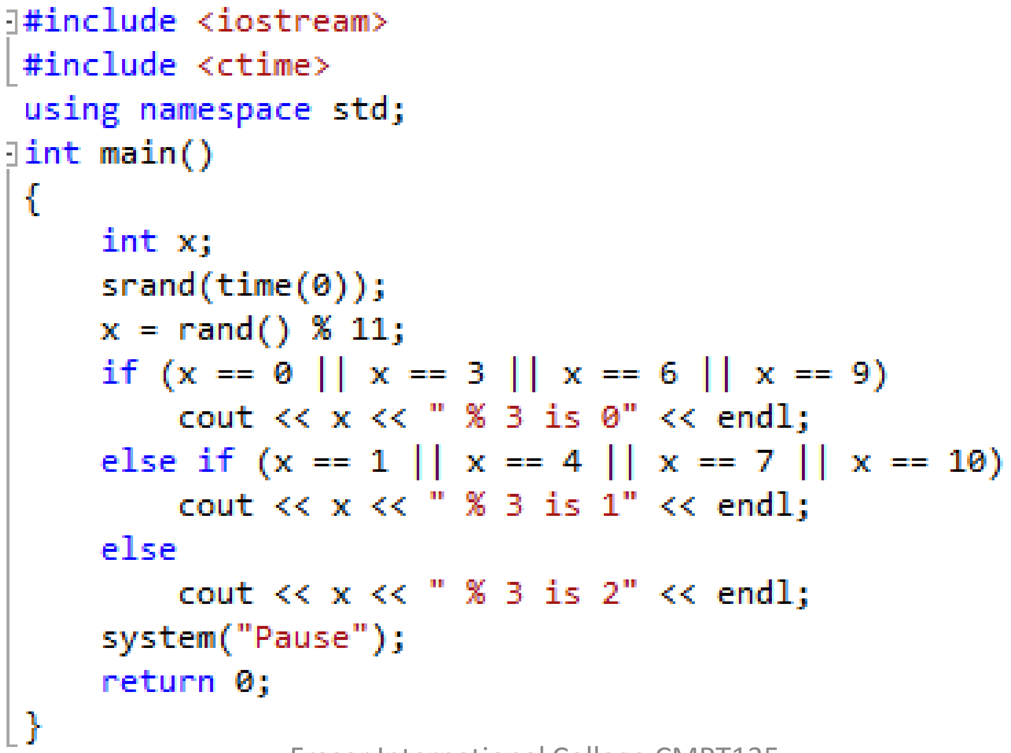
```cpp
#include <iostream>
#include <ctime>
using namespace std;
int main()
{
	int x;
	srand(time(0));
	x = rand() % 5;
	switch (x)
	{
		case 0:
			cout << "Value of x is 0" << endl;
			break;
		case 1:
			cout << "Value of x is 1" << endl;
			break;
		case 2:
			cout << "Value of x is 2" << endl;
			break;
		case 3:
			cout << "Value of x is 3" << endl;
			break;
		default:
			cout << "Value of x is 4" << endl;
	}
	system("Pause");
	return 0;
}
```

C++ switch-case statements

- The swicth-case statement checks each of the cases listed and executes the case that matches the value of the switch statement
- An important aspect of the switch-case statement is that once a matching case is found, then the matching case **and all the cases below it** will be executed
- In order to only execute the matching case, a break statement is used
- The break statement forces execution to jump to the statement below the switch-case statement

C++ switch-case statements

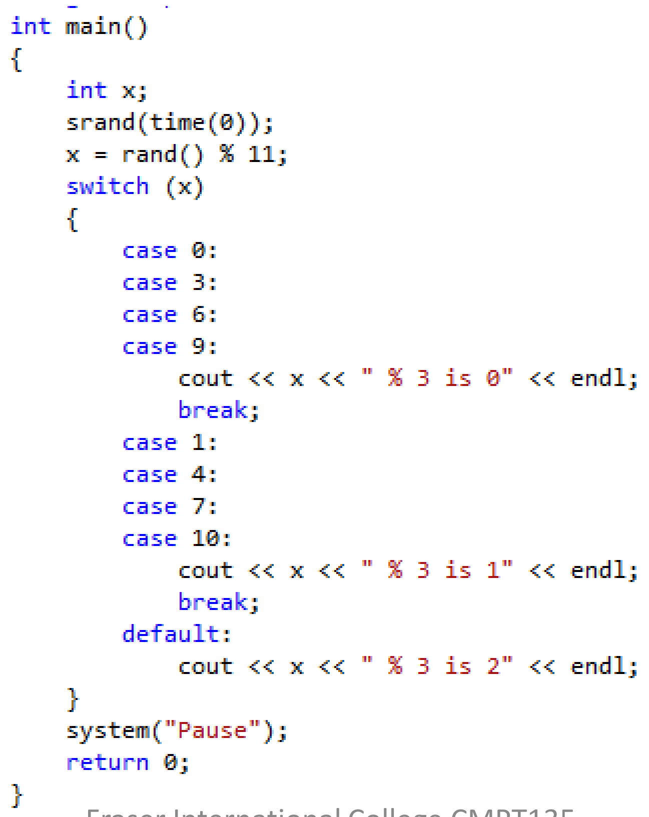
- Now consider the following program and convert the if else-if else conditional statements to an equivalent switch-case statement



```cpp
#include <iostream>
#include <ctime>
using namespace std;
int main()
{
    int x;
    srand(time(0));
    x = rand() % 11;
    if (x == 0 || x == 3 || x == 6 || x == 9)
        cout << x << " % 3 is 0" << endl;
    else if (x == 1 || x == 4 || x == 7 || x == 10)
        cout << x << " % 3 is 1" << endl;
    else
        cout << x << " % 3 is 2" << endl;
    system("Pause");
    return 0;
}
```

C++ switch-case statements

- All we need to realize is the fact that once a matching case is found, all the cases under it will be executed or until a break statement is found. Therefore the following code provides the required equivalent switch-case.



```
int main()
{
    int x;
    srand(time(0));
    x = rand() % 11;
    switch (x)
    {
        case 0:
        case 3:
        case 6:
        case 9:
            cout << x << " % 3 is 0" << endl;
            break;
        case 1:
        case 4:
        case 7:
        case 10:
            cout << x << " % 3 is 1" << endl;
            break;
        default:
            cout << x << " % 3 is 2" << endl;
    }
    system("Pause");
    return 0;
}
```