CHAPTER 2

Tasting C++

There are many ways to learn a new programming language. You can start with the basic constructs and creating your first application only after you have understood the minimal set of language features. Another trend is to start with samples and get a fundamental comprehension of the language attributes by examples. Dennis Ritchie and Brian Kernighan used this latter in their legendary book, The C Programming Language, and they started introducing C with a "Hello, World" sample. Since that time, most programming books have applied the same pattern.

You will follow a very similar pattern in this chapter. Instead of commencing with a high-level comparison of C++ and C#, you will overview a few samples to get a feeling about the common and different characteristics of these two great languages.

You will start with a "Hello, World" program, and after a short explanation you'll find yourself study a few C++ features with the Sieve of Eratosthenes algorithm that calculates prime numbers. This algorithm is a great basis to compare the performance of the two programming languages.

The chapter will conclude with a picture processing example demonstrating the amazing performance improvement you can access with C++.

In this chapter, you will learn:

- What the similar constructs are in C# and C++
- A few peculiar constructs and attributes of the C++ language
- A few performance benefits of C++

Say Hello to Have a Good Buy

Let's start with the simplest C# console application—"Hello, World" —, which was probably your first C# app that time when you were getting acquainted with the language:

C# (Program.cs)

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

This code uses the **`Main`** method of the **`Program`** class as the entry point of the application, with the **`args`** string array representing command line parameters. The **`Console`** object holds operations—including **`WriteLine`**—that can be used to send text and other data to the default output. The **`Console`** object is defined in the **`System`** namespace (and so its full name is **`System.Console`**). The **`using System`** directive at the beginning of the source declares that the compiler should use the **`System`** namespace to look up objects. The **`Program`** object is declared in the **`HelloWorld`** namespace.

Now, let's have a look at the same "Hello, World" program written in C++, using Visual Studio 2012:

C++ (HelloWorld.cpp)

```
#include "stdafx.h"
#include <iostream>

int _tmain(int argc, _TCHAR* argv[])
{
    std::cout << "Hello, World!" << std::endl;
}
```

Here, the **`_tmain`** function is the entry point of the application. It accepts two parameters, **`argc`** and **`argv`**, which represent the number of command line argument, and an array of strings holding the pieces of the command line, respectively. The **`std::cout`** object represents the console output; the **`std::endl`** object is a synonym for the line terminator character sequence. The two "**`<<`**" operators cause the "Hello, World" string and then the line terminator to be written to the console output.

The two **`#include`** directives force the compiler to process so-called header files. These files contain definitions of symbols, identifiers and operations used in the source code. Without any deeper explanation about the working details—you'll meet with header files many times in this book later—, the compiler knows the **`_tmain`** identifier from the **`"stdafx.h"`** header file defined within this application project, and the **`std::cout`** object from the **`<iostream>`** system header file.

The **std::cout** identifier is a compound name is composed from a namespace (**std**) and the name of a variable (**cout**) within that namespace.

First Impressions

Although this C++ program looks a bit esoteric for C# programmers at the first sight, it is very similar to the C# program above. Here are a few things in common:

- Both programs have clearly defined entry points with access to command line parameters.
- Both programs have a mechanism to import identifiers defined somewhere else, externally from the current source code file.
- Both programs have a concept of console output, though they seem to manage writing the output very differently.
- Both C# and C++ have the concept of namespaces.

You can also discover dissimilarities instantly:

- C# encapsulates operations (methods) into object classes, such as the **Main** method in the **Program** class, or the **WriteLine** method in the **Console** class. C++ uses its operations (functions) with being enclosed into the global scope of the application (**_tmain**).
- C++ uses the "**::**" symbol to separate a namespace from its inner object, such as in the **std::endl** expression. Although in the code snippet above you cannot explicitly see, but you know that C# uses the "." (dot) symbol to separate a namespace from its internal type, such as **System.Console** or **HelloWorld.Program**.

> ***Operations, Methods and Functions***
>
> *In their renowned book, The C Programming Language, Kernighan and Ritchie use the term* function *to describe operations independently if they returned a value or not. C and C++ programmers used to mention "*function*" with the same meaning for a long time, but also commenced to call a function with no return type a* method*. C# programmers also use* method *and* function *interchangeably. In academic circles these terms are still used to distinguish operations according to whether they return a value.*
>
> *In this book I will use the* operation, function *and* method *terms interchangeably unless it is important to use the proper term in a particular context.*

Different Concepts of Arrays

According to your C# knowledge, the meaning of **string[] args** is very clear: it represents a variable named **args**, which is an array that holds variable length Unicode strings. You also know that **args** is an object, too (it derives from **System.Object**), and so it has properties, such as **Length** or **Rank**, and operations, such as **GetValue()** and **SetValue()**, which can be accessed through **args**:

```
args.Length
args.Rank
args.GetValue(0)
args.SetValue(0, "run")
```

The **args** item is a single one in the parameter list of **Main**. If you wanted to use it, **args.Length** would tell you the number of elements in this array.

What have you seen in the equivalent C++ program? The command line parameters passed to the **_tmain** function are declared with two parameters, the **int argc**, and the **_TCHAR* argv[]**, respectively. Believe or not, in this C++ context these two parameters are equivalent with the **string[]** used in C#. Let's dive a bit deeper into the semantics of these parameter definitions!

When you define the **argv** variable as **_TCHAR* argv[]** you say that **argv** is an array that contains items with type of **_TCHAR***. An item of **_TCHAR*** is a pointer to a value of **_TCHAR**. It means that **argv** is an array of pointers to **_TCHAR** values. **_TCHAR** is a type alias for **wchar_t** that stands for a 16 bit character.

A variable in C++ is a flow of bits in the memory that represents a value. So, **argv** represents a number of consecutive bytes in the memory that represent an array of pointers to 16 bit character values. In C++, pointers and arrays are nearly related:

- An array of items with a particular type could be handled as a pointer to an item with the specific type.
- A pointer to an item with a specific type also could behave like an array of the particular type.

So, **argv** in the memory is represented as shown in Figure 2-1.

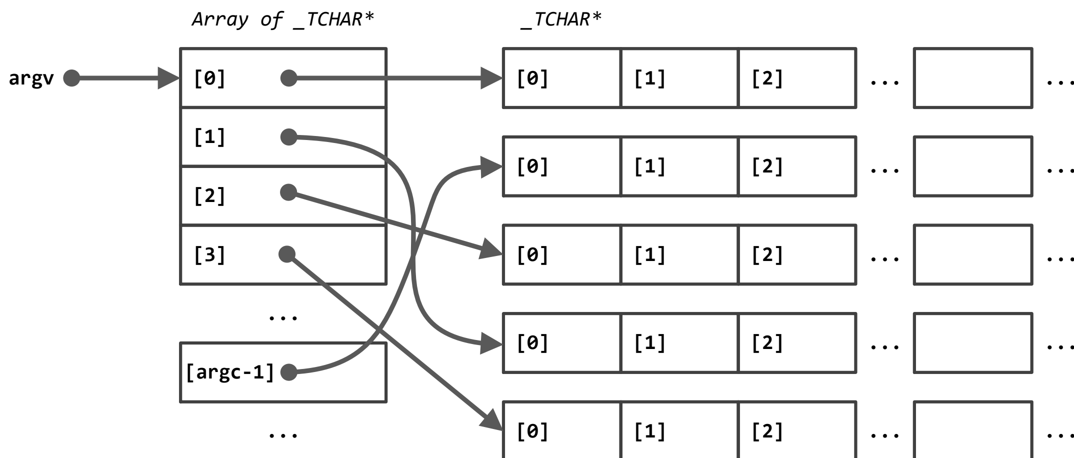

Figure 2-1: The memory representation of argv

So **argv** points to a memory location that holds a number of pointers each reserving 4 bytes of memory. The first pointer (at index 0) starts at the memory address represented by **argv**; the second one (at index 1) at **argv+4**, and so on. You may have an important question: "How many pointers do

we have in the array represented by **argv**"? The only correct answer is: "We do not know this number just by having **argv**".

The **argv** pointer (pointing to an array of **_TCHAR***) is not just like a reference to **String[]** in C#! It is simply a memory address to a consecutive set of four-byte values, each of them pointing to a memory location that represents a consecutive flow of **_TCHAR** values (each reserving 2 bytes). Recall, the **_tmain** function's first parameter is an integer value, **argc**. This value tells the number of the command line arguments, and so it is implicitly the length of the array **argv** points to.

> *Nothing prevents you to access an element of the **argv** array with an index out of bounds—by default there is no runtime check against this potential issue. It may hurt your design approach—thorough unit testing can avoid such concerns—, but the lack of this runtime check can boost performance.*

You may ask the same array length question in regard to the array of **_TCHAR** values. The elements of the **argv** array point to **_TCHAR** values—so they represent an array of **_TCHAR** values. It seems that we do not have any information about the length of this array. Well, an array of **_TCHAR** values is handled in C++ as a string, and in C++ (by design, by specification, and by convention) strings are terminated with a binary **0** character. The terminating 0 character signs the end of the array, so it implicitly determines its length.

It's great that this simple "Hello, World" application could tell a lot about C++. Now, let's move to another app that articulates a few new attributes of C++.

Prime Numbers

In this section you will take a look at an application that uses the *Sieve of Eratosthenes* to find prime numbers. If you have not heard about this algorithm yet, or you can't recall all details, here is a short summary:

Step 1: Create a list of consecutive integers from 2 to *n*: (2, 3, 4, ..., *n*).

Step 2: Initially, let *p* equal 2, the first prime number.

Step 3: Starting from *p*, count up in increments of *p* and mark each of these numbers greater than *p* itself in the list. These numbers will be 2*p*, 3*p*, 4*p*, etc.; note that some of them may have already been marked.

Step 4: Find the first number greater than *p* in the list that is not marked. If there was no such number, stop. Otherwise, let *p* now equal this number (which is the next prime), and repeat from Step 3.

*(Source: **http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes**)*

Using your C# knowledge, one possible implementation of Sieve of Eratosthenes is the one shown in Listing 2-1.

Listing 2-1: Sieve of Eratosthenes (C# implementation)

C# (FindPrimes.cs)

```
using System;
using System.Collections.Generic;

namespace FindPrimes
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> primesInAThousand = GetPrimes(1000);
            bool first = true;
            foreach(var prime in primesInAThousand)
            {
                if (!first)
                {
                    Console.Write(", ");
                }
                first = false;
                Console.Write(prime);
            }
            Console.Write("\nTotal #of primes found: {0}\n", primesInAThousand.Count);
        }

        static List<int> GetPrimes(int upperBound)
        {
            // --- Create an array of bits and initialize all of them to 1
            int length = upperBound / 8 + 1;
            var numberFlags = new byte[length];
            for (int i = 0; i < length; i++) numberFlags[i] = (byte)0xff;

            // --- Use Sieve of Eratosthenes
            int seekLimit = (int)Math.Sqrt(upperBound);
            for (int i = 2; i <= seekLimit; i++)
            {
                if ((numberFlags[i/8] & (byte)(0x80 >> i % 8)) != 0)
                {
                    for (int j = 2*i; j < upperBound; j += i)
                    {
                        numberFlags[j/8] &= (byte)(~(0x80 >> j % 8));
                    }
                }
            }

            // --- Collect prime numbers
            List<int> primeNumbers = new List<int>();
            for (int i = 2; i < upperBound; i++)
            {
```

```
                    if ((numberFlags[i/8] & (byte)(0x80 >> i % 8)) != 0)
                        primeNumbers.Add(i);
                }
                return primeNumbers;
            }
        }
    }
```

The **Main()** method calls **GetPrimes()** to retrieve the prime numbers between 2 and 1000, and then it displays the collected prime numbers on the screen. **GetPrimes()** uses the algorithm outlined earlier, with a simple technique to be frugal with the memory. Instead of using an array of Boolean values to represent marked and unmarked numbers, it uses an array of bytes (**numberFlags**) where each bit of a byte represents a number.

GetPrimes() retrieves the prime numbers in a **List<int>** instance, representing a list of integers. In the algorithm, initially all numbers are marked (they are still in the sieve):

```
for (int i = 0; i < length; i++) numberFlags[i] = (byte)0xff;
```

To check whether number ***i*** is marked, the following expression is used:

```
if ((numberFlags[i >> 3] & (byte)(0x80 >> i % 8)) != 0) { ... }
```

To unmark number ***j***, the following assignment is used:

```
numberFlags[j >> 3] &= (byte)(~(0x80 >> j % 8));
```

When you run this application, it works as expected, and it produces the output shown in Figure 2-2.

```
C:\Windows\system32\cmd.exe
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163,
 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251,
 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349,
 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443,
 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557,
 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647,
 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757,
 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863,
 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983,
 991, 997
Total #of primes found: 168
Press any key to continue . . .
```

Figure 2-2: Running the Sieve of Erathostenes to find then prime numbers between 2 and 1000

The same algorithm can be interested in C++ in a very similar way, as shown in Listing 2-2.

Listing 2-2: Sieve of Eratosthenes (C++ implementation)

C++ (FindPrimes.cpp)

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <algorithm>
#include <Windows.h>

using namespace std;

vector<int> GetPrimes(int upperBound);

int _tmain(int argc, _TCHAR* argv[])
{
    vector<int> primesInAThousand = GetPrimes(1000);
    bool first = true;
    for_each(begin(primesInAThousand), end(primesInAThousand), [&](int prime)
    {
        if (!first)
        {
            cout << ", ";
        }
        first = false;
        cout << prime;
    });
    cout << endl << "Total #of primes found: "
        << primesInAThousand.size() << endl;
      return 0;
}

vector<int> GetPrimes(int upperBound)
{
    // --- Create an array of bits and initialize all of them to 1
    int length = upperBound/8 + 1;
    BYTE* numberFlags = new unsigned char[length];
    for (int i = 0; i < length; i++) numberFlags[i] = 0xff;

    // --- Use Sieve of Eratosthenes
    int seekLimit = (int)sqrt(upperBound);
    for (int i = 2; i <= seekLimit; i++)
    {
        if (numberFlags[i/8] & (0x80 >> i % 8))
        {
            for (int j = 2*i; j < upperBound; j += i)
            {
                numberFlags[j/8] &= ~(0x80 >> j % 8);
            }
        }
    }
```

```
    // --- Collect prime numbers
    vector<int> primeNumbers;
    for (int i = 2; i < upperBound; i++)
    {
        if (numberFlags[i/8] & (0x80 >> i % 8)) primeNumbers.push_back(i);
    }
    return primeNumbers;
}
```

In this listing, **`_tmain()`** does exactly the same as **`Main()`** in the C# program, and also the C# and C++ implementations of **`GetPrimes()`** are actually identical. However, the C++ program has some interesting details; each of them tells an important attribute of the language.

The source code starts with including two headers:

```
#include <vector>
#include <algorithm>
#include <Windows.h>
```

The **`<vector>`** header contains the definition for the **`vector<>`** template, that is used as **`vector<int>`** in **`GetPrimes()`** to retrieve a collection of integers, just as **`List<int>`** does in the C# implementation. The **`<algorithm>`** header is added to be able to use the **`for_each()`** method that implements the semantics of the **`foreach`** C# statement. The **`<Windows.h>`** header contains Windows specific types and constants, it is included in order to use the **`BYTE`** type.

The implementation goes on with a using namespace directive:

```
using namespace std;
```

This directive is very similar to the **`using`** directive in C#. It tells the C++ compiler that it should look up identifiers in the **`std`** namespace. As a result, after this line you can write simply **`cout`** instead of **`std::cout`**, because the compiler will automatically look up the **`std`** namespace and it will find **`cout`** there.

The source code goes on with a declaration line:

```
vector<int> GetPrimes(int upperBound);
```

Unlike in C#, in C++ you should declare identifiers before they are used. In the source, the **`_tmain`** method's declaration precedes the declaration of **`GetPrimes()`**, but **`_tmain`** invokes **`GetPrimes()`**. Without the prior declaration of **`GetPrimes()`** the compiler would complain about an undefined identifier.

The **`_tmain()`** stores prime numbers in the **`primesInAThousand`** variable that is a **`vector<int>`**, using the **`vector<>`** template parameterized with **`int`**. A template in C++ is very similar to a C# generic

type, but at its heart, it is a very different construct. While C# generic types are a concept of the underlying .NET CLR, in C++ they are compiler-supported constructs. The **vector<>** template represents a collection that can be used for many purposes; it can be used as a dynamic list, as a queue, as a stack, and for other purposes, too. The **vector<int>** type is a vector that holds items with integer values.

The **for_each()** method implements the C# **foreach** semantics in the following form:

```
for_each(begin(primesInAThousand), end(primesInAThousand), [&](int prime)
{
    // --- Body omitted
});
```

The **begin()** and **end()** methods signify the range of items the **for_each()** method should iterate through. As the code suggests, **for_each()** goes from the first item of **primesInAThousand** to the last one. The third argument of the method call is a lambda expression. The **[&](int prime)** part defines that this expression accepts an integer value – this value is the current item of the **primesInAThousand** vector, as the loop iterates through the collection. The body of the lambda expression is closed between the braces.

Although the C# and C++ implementations of **GetPrimes()** has only slight differences, those indicate important C++ attributes.

The array of bits storing the marked/unmarked state of numbers is defined like this:

C#

```
int length = upperBound / 8 + 1;
var numberFlags = new byte[length];
```

C++

```
int length = upperBound/8 + 1;
BYTE* numberFlags = new unsigned char[length];
```

The C# implementation stores these flags in a **byte** array to leverage every bit separately as a marked/unmarked flag. However, in C++ there is no predefined **byte** data type, instead, you can use **unsigned char**. C++ allows you to use type aliases; the **BYTE** (defined in the **Windows.h** header) is such an alias for **unsigned char**. You can write any of these variable declarations due to the equivalence of **BYTE** and **unsigned char**:

```
BYTE* numberFlags = new BYTE[length];
BYTE* numberFlags = new unsigned char[length];
unsigned char* numberFlags = new unsigned char[length];
unsigned char* numberFlags = new BYTE[length];
```

The condition checking if a number is marked or unmarked is described like this:

C#

```
if ((numberFlags[i/8] & (byte)(0x80 >> i % 8)) != 0)
{
    // ...
}
```

C++

```
if (numberFlags[i/8] & (0x80 >> i % 8))
{
    // ...
}
```

The difference is that the **if** statement in C++ does not require the condition to be a Boolean expression. If the conditional expression results in a value different than 0, it is taken into account as true. In case of C# you must add "**!= 0**" to the expression in order to convert it into a Boolean expression. Another important difference is that in C++ you do not need to cast the second operand of the bitwise AND operator (&) into an **unsigned char**, unlike in C# where the second operand must be converted to **byte**.

Creating a Simple Performance Library

C++ is said to provide better performance than C#. Let's look after it! Firsts, let's create a very simple but reusable performance library that can be used to measure the execution time of an action.

In C# one of the best ways to achieve reusability is to create a standalone .NET class library with a new class that encapsulates the performance measuring functionality. The following simple class seems to be just perfect for this purpose:

C# (PerformanceTest.cs)

```
using System;
using System.Diagnostics;

namespace PerformanceLib
{
    public static class PerformanceTest
    {
        public struct PerformanceData
        {
            public int Cycles;
            public double ExecTimeInMs;
            public int Faults;
        }

        public static PerformanceData Measure(int cycles, Action action)
        {
```

```
            int faults = 0;
            var stopWatch = new Stopwatch();
            stopWatch.Start();
            for (int i = 0; i < cycles; i++)
            {
                try
                {
                    action();
                }
                catch (Exception)
                {
                    faults++;
                }
            }
            stopWatch.Stop();
            return new PerformanceData
            {
                Cycles = cycles,
                ExecTimeInMs = stopWatch.Elapsed.TotalMilliseconds,
                Faults = faults
            };
        }
    }
}
```

The **PerformanceTest** static class provides **Measure()** to check the performance of an **action** passed as the second argument of this method. The first argument of **Measure()** tells the number of cycles the same action should be executed. The results are retrieved back in an instance of the **PerformanceData** structure. The implementation uses a Stopwatch instance the carry out the measure. It is pretty straightforward, and as you can see also catches and counts exceptions.

In C++ you have several forms of reusability. When using Windows, you can compile components into a dynamic link library (DLL), and during runtime the application can load the DLL, and use the components inside. C++ also can use *static libraries* that are linked to the executable file during build time, and so they are inseparable parts of the executable. This approach is very important when you are about creating single executable files that do not depend on any external components, and can run as they are.

The C++ implementation will use the static library that defines a **PerformanceTest** class within a **PerformanceLib** namespace – using the same names as the C# implementation. The **PerformanceTest** class is defined in a header file:

C++ (PerformanceTest.h)

```
#include "stdafx.h"

namespace PerformanceLib
{
    class PerformanceTest
```

```
    {
    public:
        struct PerformaceData
        {
        public:
            int cycles;
            double execTimeInMs;
            int faults;
        };

        static PerformaceData Measure(int cycles, void action());
    };
}
```

The whole **class** definition is embedded into the **namespace** definition—similarly to the C# definition. C++ also knows the concept of **class** and **struct**; however it uses them with different semantics as C#. In the code above **PerformanceData** is nested into **PerformanceClass** and it has three public members—just like in the C# code. Notice how the **public** keyword is used here in contrast to C#. The **Measure()** method's second parameter uses an interesting notation: **void action()** means that **action** is a function that expects no argument and does not retrieve a value. As a small syntax difference between C# and C++, the latter one expects semicolons at the end of **class** and **struct** definitions.

Putting the information about **PerformanceTest** into a separate header file is necessary, because this header file will be the key of importing this definition into applications where the C++ static library will be used. Unlike C# (and other .NET languages), the artifacts of a C++ program's compilation do not expose metadata. You cannot reference a simple binary file to import its types and operations. You always need a separate header file that provides the necessary information so that the compiler can understand it.

> *Well, this situation is different with Windows 8 style applications that are built on a new component, named Windows Runtime, which provides metadata about types and operations.*

The implementation of the **PerformaceTest** class goes into a separate file:

C++ (PerformanceTest.cpp)

```
#include "stdafx.h"
#include <Windows.h>

#include "PerformanceTest.h"

namespace PerformanceLib
{
    PerformanceTest::PerformaceData PerformanceTest::Measure(int cycles, void action())
    {
```

```
        int faults = 0;
        auto start = GetTickCount64();
        for (int i = 0; i < cycles; i++)
        {
            try
            {
                action();
            }
            catch (...)
            {
                faults++;
            }
        }
        auto end = GetTickCount64();
        PerformaceData pd;
        pd.cycles = cycles;
        pd.execTimeInMs = (double)(end - start);
        pd.faults = faults;
        return pd;
    }
}
```

The file uses the #include directive to import the definitions in the header file. The implementation of **PerformanceTest** is enclosed into the **PerformanceLib** namespace. C++ does not close the implementation of operations into a separate scope explicitly: the **Measure()** method of the **PerformanceTest** class is defined as if it were an independent function. The method implementation uses the **PerformanceTest::Measure()** name and that is how the compiler knows that it is the **Measure()** method defined in the **PerformanceTest** class.

Measure() uses the **GetTickCount64()** method to measure the execution time. Observe that the **start** and **end** variables are declared with the **auto** keyword. It has the same semantics as the **var** keyword in C#. The compiler uses the initialization expression of the variable to infer its type. As a result, the type of **start** and **end** will be **ULONGLONG**—a 64-bit unsigned integer.

C++ knows the concept of structured exception handling, and manages runtime exceptions similarly to C#, with the **try…catch** construct. The **catch (…)** block traps all exceptions.

Referencing the Performance Library

Although there are only slight differences in the C# and C++ implementation of **PerformanceLib**, the way of utilizing this library in applications is pretty unlike. In C# you can create a solution, include the **PerformanceLib** project, and add a reference to a project where **PerformanceLib** is to be utilized, as shown in Figure 2-3.

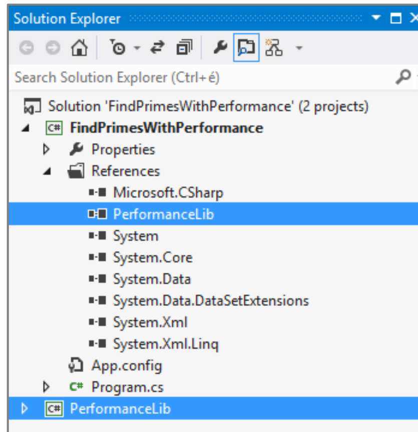

Figure 2-3: PerformanceLib added as a reference

The **`FindPrimesWithPerformance`** project can immediately use the operations of **`PerformanceLib`** in its source files.

The situation is a bit different in C++. You can still add a reference to the **`PerformaceLib`** static library, as shown in Figure 2-4, but it is not enough. You also have to add the header file with the definition of **`PerformanceLib`** to the project. You can do it by adding the header file explicitly to your project, or appending the path of the header file's folder to the Include Directories property of the project, as shown in figure 2-5. During build, the compiler will find it either because it's explicitly added, or because it's looked up in the specified include directories.

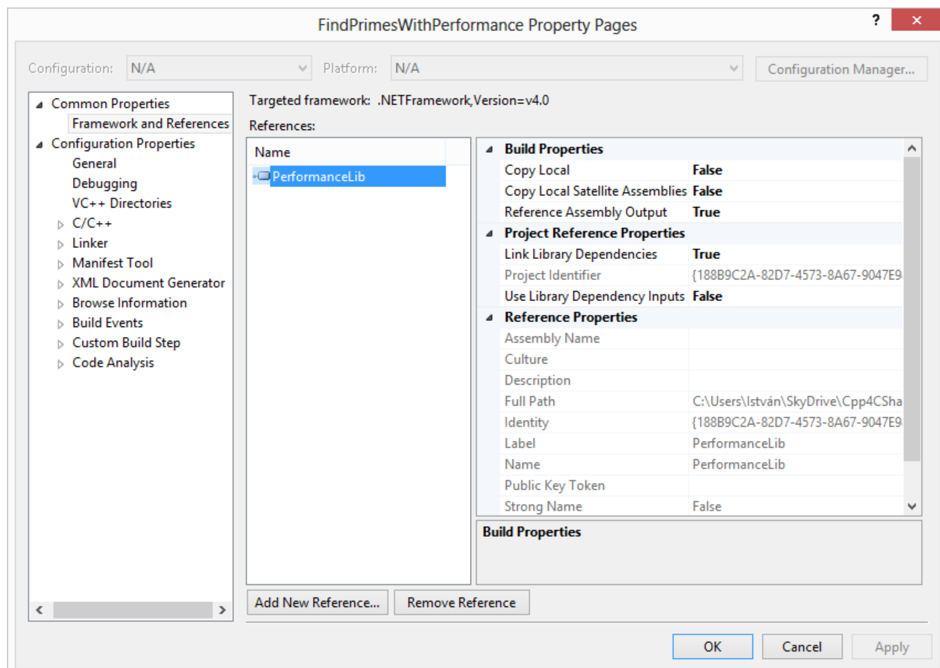

Figure 2-4: The PerformanceLib static library has to be referenced

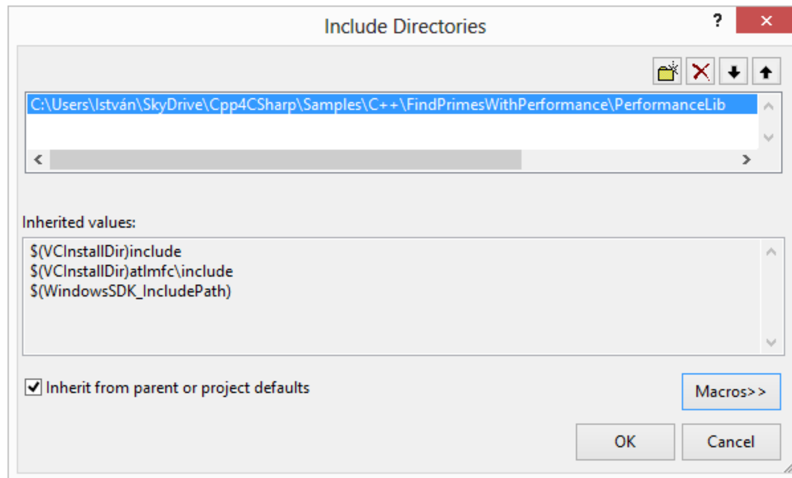

Figure 2-5: Appending the header file path to the Include Directories project property

Using the Performance Library in the Source Code

After you have added the references to **`PerformanceLib`**, it is pretty easy to utilize its operations, independently whether you work with C# or C++. In C# it is enough to add a using directive to the beginning of the file and you can immediately refer to **`PerformanceTest`**:

C# (Program.cs)

```
using System;
using System.Collections.Generic;
using PerformanceLib;

namespace FindPrimesWithPerformance
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Executing performance test...");
            var perfResult = PerformanceTest.Measure(100, () => GetPrimes(1000000));
            Console.WriteLine("Searching primes between 1 and 1,000,000 took {0} ms per call",
                perfResult.ExecTimeInMs / perfResult.Cycles);
        }

        static List<int> GetPrimes(int upperBound)
        {
            // --- Method body omitted for the sake of brevity
        }
    }
}
```

As you can see, the **GetPrimes()** method is passed to **Measure()** with a lambda expression.

In C++, you have to include the **PerformaceTest.h** header file so that the compiler recognize the **PerformanceTest** class, and optionally add a using namespace directive to access the class without explicitly using **PerformanceLib**:

C++ (FindPrimesWithPerformance.cpp)

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <algorithm>
#include "PerformanceTest.h"

using namespace std;
using namespace PerformanceLib;

vector<int> GetPrimes(int upperBound);

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "Executing performance test...\n";
    auto perfResult = PerformanceTest::Measure(100, []() {GetPrimes(1000000); });
    cout << "Searching primes between 1 and 1,000,000 took ";
    cout << perfResult.execTimeInMs / perfResult.cycles << " ms per call\n";
    return 0;
}

vector<int> GetPrimes(int upperBound)
{
    // --- Method body omitted for the sake of brevity
}
```

Just like in the C# implementation, **GetPrimes()** is passed through a lambda expression.

Performance Comparison: Sieve of Eratosthenes

Now, having **PerformanceLib** implemented both in C# and C++, we can compare the performance of the Sieve of Eratosthenes algorithm's implementations. The performance is measured by running **GetPrimes()** to find all prime numbers between 2 and 1 million, and the algorithm is run 100 times to measure the average execution time:

C#

```
var perfResult = PerformanceTest.Measure(100, () => GetPrimes(1000000));
```

C++

```
auto perfResult = PerformanceTest::Measure(100, []() {GetPrimes(1000000); });
```

We have to console apps, but to be fair during the comparison we must compile and run them in release mode. In my desktop machine having an Intel Core2 quad-core CPU running at 2.66 GHz, I measure 10.10 milliseconds for the C# app, and 8.89 milliseconds for the C++ app, as shown in Figure 2-6 and Figure 2-7.

```
C:\Windows\system32\cmd.exe
Executing performance test...
Searching primes between 1 and 1,000,000 took 10,095139 ms per call
Press any key to continue . . .
```

Figure 2-6: Running the performance test in C#

```
C:\Windows\system32\cmd.exe
Executing performance test...
Searching primes between 1 and 1,000,000 took 8.89 ms per call
Press any key to continue . . .
```

Figure 2-7: Running the performance test in C++

This result shows that the C++ app runs about 12 per cent faster. You may say, it is not significant, and in many cases I can agree with you. However, when you pay for your servers after usage, this 12 per cent really matters. For example, one of my applications running in Windows Azure costs about 1000 USD per month. In a year, 12 per cent means 12 x 1000 USD x 12 per cent that equals 1440 USD. For this amount I can spend a whole week of live-aboard scuba diving safari in Egypt—and it happens to be my favorite waste of time. So, do not underestimate the significance of the smallest performance enhancement!

Amazing Performance with C++

C++ allows you low level constructs to tune performance. When you know how the underlying CPU works, you can rephrase your code to accommodate to the CPU's feature. In the following example you will examine how this works.

The sample code contains a Windows 8 application—its name is **`GrayScaleOperation`**—that converts a color image into a grayscale one. This app also allows you to measure the performance of the conversion. When you run, it provides a very simple user interface, as shown in Figure 2-8.

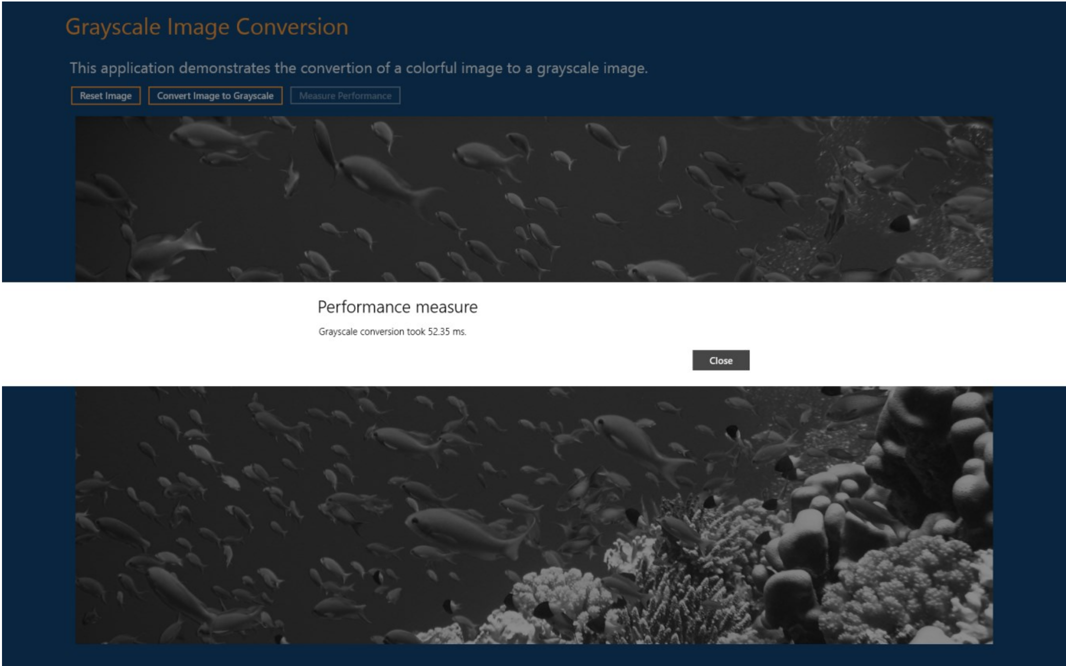

Figure 2-8: The **`GrayScaleOperation`** sample app in action

The essence of this application is the **`BitmapProcessor`** class that carries out the conversion. The image is represented in the memory as an array of bytes. Each pixel is described by four bytes: the blue, green, and red components of the pixel, and the alpha channel (opacity) value. The first pixel is the one in the top-left corner, and the subsequent one is the pixel in the same row and the next column, and so on. After the last pixel of the first row, the subsequent one is the first pixel of the next row, and so on till the last pixel of the image. The memory layout of the image is shown in Figure 2-9.

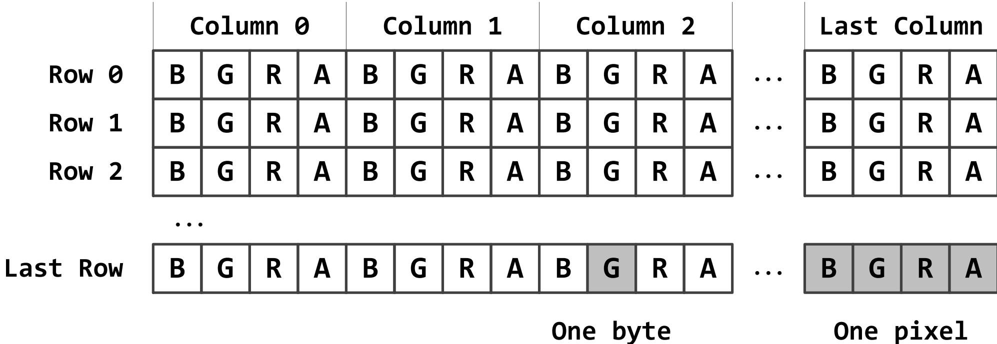

Figure 2-9: The layout of the image in the memory

The grayscale conversion means that after the conversion each of the blue (**`B`**), green (**`G`**) and red (**`R`**) components of the color are set to the same value (**`GV`**) that is calculated with the following formula:

```
GV = 0.11*B + 0.59*G + 0.3*R
```

The C# implementation transforming the array of bytes in the image is the following:

C# (BitmapProcessor.cs)

```
public static class BitmapProcessor
{
    public static void ConvertToGrayScale(byte[] sourcePixels, int width, int height)
    {
        for (int i = 0, j = 0; j < width * height; j++, i += 4)
        {
            var grayValue = (byte)(0.11f * sourcePixels[i] + 0.59f * sourcePixels[i + 1]
                + 0.3f * sourcePixels[i + 2]);
            sourcePixels[i] = sourcePixels[i + 1] = sourcePixels[i + 2] = grayValue;
        }
    }
}
```

The analogous C++ implementation is straightforward:

C++ (BitmapProcessor.cpp) – First version

```
void BitmapProcessor::ConvertToGrayScale(byte* sourcePixels, int width, int height)
{
        for (int i = 0, j = 0; j < width * height; j++, i += 4)
        {
            auto grayValue = (byte)(0.11f * sourcePixels[i] + 0.59f * sourcePixels[i + 1]
                + 0.3f * sourcePixels[i + 2]);
            sourcePixels[i] = sourcePixels[i + 1] = sourcePixels[i + 2] = grayValue;
        }
}
```

Tuning with low level constructs

Knowing the fact that C++ source code is compiled directly to CPU instructions, and the fact that the CPU can work efficiently with pointers and direct addressing of the memory—the algorithm can be written in a more verbose, but hopefully faster way:

C++ (BitmapProcessor.cpp) – Optimized version

```
void BitmapProcessor::ConvertToGrayScaleWithOptimization(byte* source, int width, int height)
{
    for (int i = 0; i < width * height; i++)
    {
        float grayValue = *source * 0.11f;
        source++;
        grayValue += *source * 0.59f;
        source++;
        grayValue += *source * 0.3f;
        byte gray = (byte)grayValue;
        *source = gray;
        source--;
        *source = gray;
        source--;
        *source = gray;
        source += 4;
    }
}
```

In this code **source** is used as a pointer to a consecutive set of bytes—as it has been mentioned earlier, in C++ array names can be used as pointers and vice versa—, and ***source** is the byte **source** is currently pointing to. The **source++** operation increments the pointer—so after this operation **source** points to the next byte—, meanwhile **source--** decrements the pointer and it moves back to the previous byte. The **source += 4** assignment moves **source** with 4 bytes—so if **source** pointed to the B color channel of a pixel before, after it would point to the B channel of the next pixel. You can check that this C++ code does exactly the same as the previous version, even if it is a bit more difficult to recognize.

Measuring the Performance

Now, you have three versions to carry out the same operation. Let's compare their performance! The following table shows the findings:

	C#	C++	C++ Low Level
Execution time (milliseconds)	38.73	19.03	**17.94**
Execution time compared to **C++ Low Level** (%)	196.20	106.08	**100.00**

It is not surprising that the C++ code is almost twice as fast as the C# code. But the table also reflects that a simple tuning in the C++ code—that uses exactly the same operation with low level constructs—is about six per cent faster!

Of course, anyone can discuss whether this six per cent is worth such a tuning. In most cases, it does not, because developer hours cost much more than the return of such a tuning. However, there may

be situations when this six percent lets you perform better than your competitor and might mean a win in a contention.

Summary

As you learned in this chapter, C++ and C# share a number of important concepts, such as

- they both use namespaces;
- they both have a mechanism to import identifiers defined somewhere else, externally from the current source code file;
- they both have a concept of console output

Well, C++ has its own unique features that allow you to write constructs with control closer to the hardware. While in C# you work with objects (in .NET even values, such as integers and strings are objects) that inherit directly or indirectly from the root `System.Object` type, C++ works with values. Operations in C# are always encapsulated in types; meanwhile C++ uses functions logically encapsulated into the global scope of the application.

Arrays in C++ are a great example of being close to the hardware. While C# arrays are objects with many predefined operations that check the array boundaries, in C++ an array is just a pointer to a consecutive block of memory with the items of the array, and without an explicit length. There are no boundary checks carried out during runtime (you can ask the compiler to do it, but it's turned off by default).

The lower level constructs of C++ provide you great performance. In many cases C# is just a little behind the performance of C++, but there are circumstances where C++ is much faster. The simple concepts in C++—such as pointers and pointer arithmetic—can help you to tune your program's performance further.

Samples Used in This Chapter

Sample Name	Description
HelloWorld	The implementation of the simplest "Hello, World" program
FindPrimes	The simple console application finding prime numbers
FindPrimesWithPerformance	The **FindPrimes** application with a simple **PerformanceLib** added, which measures the performance of finding prime numbers
GrayScaleOperation	A Windows 8 application that converts a color image to a grayscale image. The C++ version implements two versions of the grayscale conversion.