Quick C#

Aisha Ikram, 17 Jun 2003

Learn C# in less than an hour. Discover the C# language constructs and features in a brief yet comprehensive way using code examples. This article is especially good if you know C++ and feel lazy about learning C#!

Introduction

C# is a language with the features of C++, programming style like Java and rapid application model of BASIC. If you already know the C++ language, it will take you less than an hour to quickly go through the syntax of C#. Familiarity with Java will be a plus, as Java program structure, the concept of packages and garbage collection will definitely help you learn C# more quickly. So while discussing C# language constructs, I will assume, you know C++.

This article discusses the C# language constructs and features using code examples, in a brief and comprehensive way, so that you just by having a glance at the code, can understand the concepts.

Note: This article is not for C# gurus. There must be some other beginner's articles on C#, but this is yet another one.

Following topics of C# language are discussed:

- Program structure
- Namespaces
- Data types
- Variables
- Operators and expressions
- Enumerations
- Statements
- Classes and structs
- Modifiers
- Properties
- Interfaces
- Function parameters
- Arrays
- Indexers
- Boxing and unboxing
- Delegates
- Inheritance and polymorphism

Following are not discussed:

- Things which are common in C++ and C#.
- Concepts like garbage collection, threading, file processing etc.
- Data type conversions
- Exception handling
- .NET library

Program structure

Like C++, C# is case-sensitive. Semi colon (`;`) is the statement separator. Unlike C++, there are no separate declaration (header) and implementation (CPP) files in C#. All code (class declaration and implementation) is placed in one file with extension *cs*.

Have a look at this Hello world program in C#.

Hide Copy Code

```
using System;

namespace MyNameSpace

{

class HelloWorld

{
    static void Main(string[] args)
    {
        Console.WriteLine ("Hello World");
     }
}

}
```

Everything in C# is packed into a class and classes in C# are packed into namespaces (just like files in a folder). Like C++, a main method is the entry point of your program. C++'s main function is called `main` whereas C#'s main function starts with capital M and is named as `Main`.

No need to put a semi colon after a class block or `struct` definition. It was in C++, C# doesn't require that.

Namespace

Every class is packaged into a namespace. Namespaces are exactly the same concept as in C++, but in C# we use namespaces more frequently than in C++. You can access a class in a namespace using dot (`.`) qualifier. `MyNameSpace` is the namespace in hello world program above.

Now consider you want to access the `HelloWorld` class from some other class in some other namespace.

Hide Copy Code

```
using System;
namespace AnotherNameSpace
{
    class AnotherClass
    {
        public void Func()
        {
            Console.WriteLine ("Hello World");
        }
    }
}
```

Now from your `HelloWorld` class you can access it as:

Hide Copy Code

```
using System;
using AnotherNameSpace; // you will add this using statement
namespace MyNameSpace
{
class HelloWorld
{
    static void Main(string[] args)
    {
        AnotherClass obj = new AnotherClass();
        obj.Func();
    }
}
}
```

In .NET library, `System` is the top level namespace in which other namespaces exist. By default there exists a global namespace, so a class defined outside a namespace goes directly into this global namespace and hence you can access this class without any qualifier.

You can also define nested namespaces.

Using

The `#include` directive is replaced with `using` keyword, which is followed by a namespace name. Just as `using System` as above. `System` is the base level namespace in which all other namespaces and classes are packed. The base class for all objects is `Object` in the `System` namespace.

Variables

Variables in C# are almost the same as in C++ except for these differences:

1. Variables in C# (unlike C++), always need to be initialized before you access them, otherwise you will get compile time error. Hence, it's impossible to access an un-initialized variable.
2. You can't access a “dangling” pointer in C#.
3. An expression that indexes an array beyond its bounds is also not accessible.
4. There are **no global variables** or functions in C# and the behavior of globals is achieved through static functions and static variables.

Data types

All types of C# are derived from a base class `object`. There are two types of data types:

1. Basic/ built-in types
2. User-defined types

Following is a table which lists built-in C# types:

Type	Bytes	Description
`byte`	1	unsigned byte
`sbyte`	1	signed byte
`short`	2	signed short
`ushort`	2	unsigned short
`int`	4	signed integer
`uint`	4	unsigned integer
`long`	8	signed long
`ulong`	8	unsigned long
`float`	4	floating point number

`double`	8	double precision number
`decimal`	8	fixed precision number
`string`		Unicode string
`char`		Unicode char
`bool`	`true`, `false`	boolean

Note: Type range in C# and C++ are different, example, long in C++ is 4 bytes, and in C# it is 8 bytes. Also the `bool` *and* `string` *types are different than those in C++.* `bool` *accepts only* `true` *and* `false` *and not any integer.*

User defined types includes:

1. `Class`es
2. `Struct`s
3. `Interface`s

Memory allocation of the data types divides them into two types:

1. Value types
2. Reference types

Value types

Values types are those data types which are allocated in stack. They include:

- All basic or built-in types except `string`s
- `Struct`s
- `Enum` types

Reference types

Reference types are allocated on heap and are garbage collected when they are no longer being used. They are created using `new` operator, and there is no `delete` operator for these types unlike C++ where user has to explicitly delete the types created using `delete` operator. In C#, they are automatically collected by garbage collector.

Reference types include:

- `Class`es
- `Interface`s
- Collection types like `Array`s
- `String`

Enumeration

Enumerations in C# are exactly like C++. Defined through a keyword `enum`.

Example:

Hide Copy Code

```
enum Weekdays
{
    Saturday, Sunday, Monday, Tuesday, Wednesday, Thursday, Friday
}
```

Classes and structs

`Class`es and `struct`s are same as in C++, except the difference of their memory allocation. Objects of classes are allocated

in heap, and are created using `new`, where as `struct`s are allocated in stack. `Struct`s in C# are very light and fast data types. For heavy data types, you should create classes.

Examples:

Hide Shrink Copy Code

```
struct Date
{
    int day;
    int month;
    int year;
}

class Date
{
    int day;
    int month;
    int year;
    string weekday;
    string monthName;
    public int GetDay()
    {
        return day;
    }
    public int GetMonth()
    {
        return month;
    }
    public int GetYear()
    {
        return year;
    }
    public void SetDay(int Day)
    {
        day = Day ;
    }
    public void SetMonth(int Month)
    {
        month = Month;
    }
    public void SetYear(int Year)
    {
        year = Year;
    }
    public bool IsLeapYear()
    {
        return (year/4 == 0);
    }
    public void SetDate (int day, int month, int year)
    {
    }
    ...
}
```

Properties

If you are familiar with the object oriented way of C++, you must have an idea of properties. Properties in above example of `Date` class are `day`, `month` and `year` for which in C++, you write `Get` and `Set` methods. C# provides a more convenient, simple and straight forward way of accessing properties.

So above class can be written as:

Hide Shrink Copy Code

```
using System;
class Date
```

```
{
    public int Day{
        get {
            return day;
        }
        set {
            day = value;
        }
    }
    int day;

    public int Month{
        get {
            return month;
        }
        set {
            month = value;
        }
    }
    int month;

    public int Year{
        get {
            return year;
        }
        set {
            year = value;
        }
    }
    int year;

    public bool IsLeapYear(int year)
    {
        return year%4== 0 ? true: false;
    }
    public void SetDate (int day, int month, int year)
    {
        this.day = day;
        this.month = month;
        this.year = year;
    }
}

Here is the way you will get and set these properties:
class User
{
   public static void Main()
   {
        Date date = new Date();
        date.Day = 27;
        date.Month = 6;
        date.Year = 2003;
        Console.WriteLine
         ("Date: {0}/{1}/{2}", date.Day, date.Month, date.Year);
    }
}
```

Modifiers

You must be aware of `public`, `private` and `protected` modifiers that are commonly used in C++. I will here discuss some new modifiers introduced by C#.

readonly

`readonly` modifier is used only for the class data members. As the name indicates, the `readonly` data members can only

be read, once they are written either by directly initializing them or assigning values to them in constructor. The difference between the `readonly` and `const` data members is that `const` requires you to initialize with the declaration, that is directly. See example code:

Hide Copy Code

```
class MyClass
{
    const int constInt = 100; //directly
    readonly int myInt = 5; //directly
    readonly int myInt2;

    public MyClass()
    {
        myInt2 = 8;        //Indirectly
    }
    public Func()
    {
        myInt = 7; //Illegal
        Console.WriteLine(myInt2.ToString());
    }

}
```

sealed

`sealed` modifier with a class don't let you derive any class from it. So you use this `sealed` keyword for the classes which you don't want to be inherited from.

Hide Copy Code

```
sealed class CanNotbeTheParent
{
    int a = 5;
}
```

unsafe

You can define an unsafe context in C# using `unsafe` modifier. In unsafe context, you can write an unsafe code, example: C++ pointers etc. See the following code:

Hide Copy Code

```
public unsafe MyFunction( int * pInt, double* pDouble)
{
    int* pAnotherInt = new int;
    *pAnotherInt  = 10;
    pInt = pAnotherInt;
    ...
    *pDouble = 8.9;
}
```

Interfaces

If you have an idea of COM, you will immediately know what I am talking about. An `interface` is the abstract base class containing only the function signatures whose implementation is provided by the child class. In C#, you define such classes as interfaces using the `interface` keyword. .NET is based on such interfaces. In C#, where you can't use multiple class inheritance, which was previously allowed in C++, the essence of multiple inheritance is achieved through interfaces. That's your child class may implement multiple interfaces.

Hide Shrink Copy Code

```
using System;
```

```
interface myDrawing
{
    int originx
    {
        get;
        set;
    }
    int originy
    {
        get;
        set;
    }
    void Draw(object shape);
}

class Shape: myDrawing
{
    int OriX;
    int OriY;

    public int originx
    {
        get{
            return OriX;
        }
        set{
            OriX = value;
        }
    }
    public int originy
    {
        get{
            return OriY;
        }
        set{
            OriY = value;
        }
    }
    public void Draw(object shape)
    {
        ... // do something
    }

    // class's own method
    public void MoveShape(int newX, int newY)
    {
    .....
    }

}
```

Arrays

Arrays in C# are much better than C++. Arrays are allocated in heap and thus are reference types. You can't access an out of bound element in an array. So C# prevents you from that type of bugs. Also some helper functions to iterate array elements are provided. `foreach` is the statement for such iteration. The difference between the syntax of C++ and C# array is:

- The square brackets are placed after the type and not after the variable name
- You create element locations using `new` operator.

C# supports single dimensional, multi dimensional, and ***jagged arrays*** (array of array).

Examples:

Hide Copy Code

```
int[] array = new int[10]; // single-dimensional array of int
```

```
for (int i = 0; i < array.Length; i++)
    array[i] = i;

int[,] array2 = new int[5,10]; // 2-dimensional array of int
array2[1,2] = 5;

int[,,] array3 = new int[5,10,5]; // 3-dimensional array of int
array3[0,2,4] = 9;

int[][] arrayOfarray = new int[2]; // Jagged array - array of array of int
arrayOfarray[0] = new int[4];
arrayOfarray[0] = new int[] {1,2,15};
```

Indexers

Indexer is used to write a method to access an element from a collection, by straight way of using `[]`, like an array. All you need is to specify the index to access an instance or element. Syntax of Indexer is same as that of class properties, except they take the input parameter, that is the index of the element.

Example:

Note: `CollectionBase` *is the library class used for making collections.* `List` *is the protected member of* `CollectionBase` *which stores the collection list.*

Hide Copy Code

```
class Shapes: CollectionBase
{
    public void add(Shape shp)
    {
        List.Add(shp);
    }

    //indexer
    public Shape this[int index]
    {
        get {
            return (Shape) List[index];
        }
        set {
            List[index] = value ;
         }
     }
}
```

Boxing/Unboxing

The idea of boxing is new in C#. As mentioned above, all data types, built-in or user defined, are derived from a base class `object` in the `System` namespace. So the packing of basic or primitive type into an `object` is called *boxing*, whereas the reverse of this known as *unboxing*.

Example:

Hide Copy Code

```
class Test
{
   static void Main()
   {
      int myInt = 12;
      object obj = myInt ;      // boxing
      int myInt2 = (int) obj;   // unboxing
   }
```

```
}
```

Example shows both boxing and unboxing. An `int` value can be converted to `object` and back again to `int`. When a variable of a value type needs to be converted to a reference type, an object box is allocated to hold the value, and the value is copied into the box. Unboxing is just the opposite. When an object box is cast back to its original value type, the value is copied out of the box and into the appropriate storage location.

Function parameters

Parameters in C# are of three types:

1. By-Value/In parameters
2. By-Reference/In-Out parameters
3. Out parameters

If you have an idea of COM interface and it's parameters types, you will easily understand the C# parameter types.

By-Value/In parameters

The concept of value parameters is same as in C++. The value of the passed value is copied into a location and is passed to the function.

Example:

Hide Copy Code

```
SetDay(5);
...
void SetDay(int day)
{
    ....
}
```

By-Reference/In-Out parameters

The reference parameters in C++ are passed either through pointers or reference operator `&`. In C# reference parameters are less error prone. Reference parameters are also called In-Out parameters because you pass a reference address of the location, so you pass an input value and get an output value from that function.

You can not pass an un-initialized reference parameter into a function. C# uses a keyword `ref` for the reference parameters. You also have to use keyword `ref` with an argument while passing it to a function demanding reference parameter.

Example:

Hide Copy Code

```
int a= 5;
FunctionA(ref a); // use ref with argument or you will get compiler error
Console.WriteLine(a); // prints 20
```

Hide Copy Code

```
void FunctionA(ref int Val)
{
    int x= Val;
    Val = x* 4;
}
```

Out parameter

Out parameter is the parameter which only returns value from the function. The input value is not required. C# uses a keyword

`out` for the out parameters

Example:

Hide Copy Code

```
int Val;
GetNodeValue(Val);
```

Hide Copy Code

```
bool GetNodeValue(out int Val)
{
    Val = value;
    return true;
}
```

Variable number of parameters and arrays

Arrays in C# are passed through a keyword `params`. An array type parameter should always be the right most argument of the function. Only one parameter can be of array type. You can pass any number of elements as an argument of type of that array. You can better understand it from example below:

Note: This is the only way C# provides for optional or variable number of parameters, that is using array.

Example:

Hide Copy Code

```
void Func(params int[] array)
{
    Console.WriteLine("number of elements {0}", array.Length);
}
```

Hide Copy Code

```
Func(); // prints 0
Func(5); // prints 1
Func(7,9); // prints 2
Func(new int[] {3,8,10}); // prints 3
int[] array = new int[8] {1,3,4,5,5,6,7,5};
Func(array); // prints 8
```

Operators and expressions

Operators are exactly the same as of C++ and thus the expression also. However some new and useful operators are also added. Some of them are discussed here.

is operator

`is` operator is used to check whether the operand types are equal or convert-able. The `is` operator is particularly useful in the polymorphism scenarios. `is` operator takes two operands and the result is a boolean. See the example:

Hide Copy Code

```
void function(object param)
{
    if(param is ClassA)
        //do something
    else if(param is MyStruct)
        //do something
    }
}
```

as operator

`as` operator checks if the type of the operands are convert-able or equal (as is done by `is` operator) and if it is, the result is a converted or boxed object (if the operand can be boxed into the target type, see boxing/unboxing). If the objects are not convert-able or box-able, the return is a `null`. Have a look at the example below to better understand the concept.

Hide Copy Code

```
Shape shp = new Shape();
Vehicle veh = shp as Vehicle; // result is null, types are not convertable

Circle cir = new Circle();
Shape shp = cir;
Circle cir2 = shp as Circle;  //will be converted

object[] objects = new object[2];
objects[0] = "Aisha";
object[1] = new Shape();

string str;
for(int i=0; i&< objects.Length; i++)
{
    str = objects[i] as string;
    if(str == null)
        Console.WriteLine("can not be converted");
    else
        Console.WriteLine("{0}",str);
}

Output:
Aisha
can not be converted
```

Statements

Statements in C# are just like in C++ except some additions of new statements and modifications in some statements.

Followings are new statements:

foreach

For iteration of collections like arrays etc.

Example:

Hide Copy Code

```
foreach (string s in array)
    Console.WriteLine(s);
```

lock

Used in threads for locking a block of code making it a critical section.

checked/unchecked

The statements are for overflow checking in numeric operations.

Example:

Hide Copy Code

```
int x = Int32.MaxValue; x++; // Overflow checked
{
x++; // Exception
}
unchecked
{
x++; // Overflow}
}
Following statements are modified:
```

Switch

`Switch` statement is modified in C#.

1. Now after executing a `case` statement, program flow can not jump to next case which was previously allowed in C++.

   Example:

   Hide Copy Code

   ```
   int var = 100;
   switch (var)
   {
       case 100: Console.WriteLine("<Value is 100>"); // No break here
       case 200: Console.WriteLine("<Value is 200>"); break;
   }
   ```

   Output in C++:

   Hide Copy Code

   ```
   <Value is 100><Value is 200>
   ```

   In C# you get compile time error:

   Hide Copy Code

   ```
   error CS0163: Control cannot fall through
          from one case label ('case 100:') to another
   ```

2. However you can do this similar to how you do it in C++:

   Hide Copy Code

   ```
   switch (var)
   {
       case 100:
       case 200: Console.WriteLine("100 or 200<VALUE is 200>"); break;
   }
   ```

3. You can also use constant variables for case values:

   Example:

   Hide Copy Code

   ```
   const string WeekEnd = "Sunday";
   const string WeekDay1 = "Monday";

   ....

   string WeekDay = Console.ReadLine();
   switch (WeekDay )
   {
   case WeekEnd: Console.WriteLine("It's weekend!!"); break;
   case WeekDay1: Console.WriteLine("It's Monday"); break;

   }
   ```

Delegates

Delegates let us store function references into a variable. In C++, this is like using and storing function pointer for which we usually use `typedef`.

Delegates are declared using a keyword `delegate`. Have a look at this example, and you will understand what delegates are:

Example:

Hide Shrink Copy Code

```
delegate int Operation(int val1, int val2);
public int Add(int val1, int val2)
{
    return val1 + val2;
}
public int Subtract (int val1, int val2)
{
    return val1- val2;
}

public void Perform()
{
    Operation Oper;
    Console.WriteLine("Enter + or - ");
    string optor = Console.ReadLine();
    Console.WriteLine("Enter 2 operands");

    string opnd1 = Console.ReadLine();
    string opnd2 = Console.ReadLine();

    int val1 = Convert.ToInt32 (opnd1);
    int val2 = Convert.ToInt32 (opnd2);

    if (optor == "+")
        Oper = new Operation(Add);
    else
        Oper = new Operation(Subtract);

    Console.WriteLine(" Result = {0}", Oper(val1, val2));
}
```

Inheritance and polymorphism

Only single inheritance is allowed in C#. Multiple inheritance can be achieved using interfaces.

Example:

Hide Copy Code

```
class Parent{
}

class Child : Parent
```

Virtual functions

Virtual functions to implement the concept of polymorphism are same in C#, except you use the `override` keyword with the virtual function implementation in the child class. The parent class uses the same `virtual` keyword. Every class which overrides the virtual method will use `override` keyword.

Hide Shrink Copy Code

```
class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Shape.Draw")    ;
    }
}

class Rectangle : Shape

{
    public override void Draw()
    {
        Console.WriteLine("Rectangle.Draw");
    }
}

class Square : Rectangle
{
    public override void Draw()
    {
        Console.WriteLine("Square.Draw");
    }
}
class MainClass
{
    static void Main(string[] args)
    {
        Shape[] shp = new Shape[3];
        Rectangle rect = new Rectangle();

        shp[0] = new Shape();
        shp[1] = rect;
        shp[2] = new Square();

        shp[0].Draw();
        shp[1].Draw();
        shp[2].Draw();
    }
}
Output:
Shape.Draw
Rectangle.Draw
Square.Draw
```

Hiding parent functions using "new"

You can define in a child class a new version of a function, hiding the one which is in base class. A keyword `new` is used to define a new version. Consider the example below, which is a modified version of above example and note the output this time, when I replace the keyword `override` with a keyword `new` in `Rectangle` class.

Hide Shrink Copy Code

```
class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Shape.Draw")    ;
    }
}

class Rectangle : Shape
{
    public new void Draw()
    {
        Console.WriteLine("Rectangle.Draw");
```

```
    }
}
class Square : Rectangle
{
    //wouldn't let u override it here
    public new void Draw()
    {
        Console.WriteLine("Square.Draw");
    }
}
class MainClass
{
    static void Main(string[] args)
    {
        Console.WriteLine("Using Polymorphism:");
        Shape[] shp = new Shape[3];
        Rectangle rect = new Rectangle();

        shp[0] = new Shape();
        shp[1] = rect;
        shp[2] = new Square();

        shp[0].Draw();
        shp[1].Draw();
        shp[2].Draw();

        Console.WriteLine("Using without Polymorphism:");
        rect.Draw();
        Square sqr = new Square();
        sqr.Draw();
    }
}

Output:
Using Polymorphism
Shape.Draw
Shape.Draw
Shape.Draw
Using without Polymorphism:
Rectangle.Draw
Square.Draw
```

See how the polymorphism doesn't take the `Rectangle` class's `Draw` method as a polymorphic form of the `Shape`'s `Draw` method, instead it considers it a different method. So in order to avoid the naming conflict between parent and child, we have used `new` modifier.

Note: you can not use in the same class the two versions of a method, one with `new` *modifier and other with* `override` *or* `virtual`*. Like in above example, I can not add another method named* `Draw` *in* `Rectangle` *class which is a* `virtual` *or* `override` *method. Also in the* `Square` *class, I can't override the virtual* `Draw` *method of* `Shape` *class.*

Calling base class members

If the child class has the data members with same name as that of base class, in order to avoid naming conflicts, base class data members and functions are accessed using a keyword `base`. See in examples how the base class constructors are called and how the data members are used.

Hide Copy Code

```
public Child(int val) :base(val)
{
    myVar = 5;
    base.myVar;
}

OR
```

```
public Child(int val)
{
    base(val);
    myVar = 5 ;
    base.myVar;
}
```

Future additions

This article is just a quick overview of the C# language so that you can just become familiar with the language features. Although I have tried to discuss almost all the major concepts in C# in a brief and comprehensive way with code examples, yet I think there is lot much to be added and discussed.

In future, I would like to add more commands and concepts not yet discussed, including events etc. I would also like to write for beginners, about Windows programming using C#.

References:

- Our most commonly known MSDN
- Inside C# by Tom Archer
- A Programmer's Introduction to C# by Eric Gunnerson
- Beginning C# by Karli Watson
- Programming C# (O'Reilly)

Modifications:

- June 12, 2003: By-Reference/In-Out Parameters- added `ref` keyword while calling a function with reference parameters
- June 20, 2003: Added a note for the optional parameters, corrected typo mistake of assignment operator in example of jagged array

License

Share

About the Author

Aisha Ikram
Technical Lead
United Kingdom

I am currently working as a Technical Lead in UK. I worked in .NET 1.1/2.0, C#, VB.NET, ASP.NET, VC++ 6, MFC, ATL, COM/DCOM, SQL Server 2000/2005. These days i am learning all .Net 3.x stuff. My free source code and articles website at http://aishai.netfirms.com

You may also be interested in...


Regex Quick Reference


Generate and add keyword variations using AdWords API


Quick Print


Window Tabs (WndTabs) Add-In for DevStudio


SAPrefs - Netscape-like Preferences Dialog


OLE DB - First steps

Comments and Discussions

You must Sign In to use this message board.

Search Comments Go

Spacing Relaxed Layout Normal Per page 25 Update

First Prev Next

Message Closed	Member 12075599	24-Nov-16 3:17
Message Closed	Member 12075599	24-Nov-16 3:14
Message Closed	Member 12075599	23-Nov-16 23:06
Message Closed	Member 12075599	20-Nov-16 21:45

Message Closed	Member 12075599	20-Nov-16 21:43
Message Closed	Member 12075599	20-Nov-16 21:33
Thanks everyone	Aisha Ikram	18-Nov-16 10:53
My vote of 4	Itz.Irshad	28-Sep-16 23:37
Good	Gurtej kanwar	17-Feb-16 0:23
For newbie Learning	Ehtesham Astute	15-Feb-16 17:38
good	mayank yadav	14-Jun-15 7:38
Should I read, or too outdated ??	Member 9986011	4-Jun-15 14:50
Great Introduction to C#	Vassilis Tsolekas	1-Oct-14 1:28
My vote of 4	Sibeesh Venu	7-Aug-14 19:59
It's very good.	Youngman Park	2-Apr-14 18:29
Thank you everyone	Aisha Ikram	11-Aug-13 23:15
My vote of 5	Amir Mohammad Nasrollahi	11-Aug-13 20:17
My vote of 5	Heaven2020	18-Mar-13 3:05
Excellent	Mohsin Azam	8-Dec-12 6:14
My vote of 5	OMAR SALIM	3-May-12 1:33
Quick C	avatardeath	11-Mar-12 6:00
It is easy to understand.	bylion1990	21-Feb-12 4:19
My vote of 5	ganeshaditya310	9-Jan-12 19:19
My vote of 4	Soul Harvester	21-Oct-11 0:09
My vote of 4	rok patel	28-Aug-11 22:36

Last Visit: 31-Dec-99 18:00 Last Update: 30-Jul-17 19:59 Refresh 1 2 3 4 5 Next »

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.