5. The C# Language and System

This chapter, together with Chapter 6, Chapter 7, and Chapter 9, is an introduction to the C# language and the C# system. On Windows, the latter is known as .Net. On purpose, we will keep the .Net part of the material very short. Our main interest in this lecture is how to program in C#, and how this is related to programming in other languages such as C, Java, and Visual Basic.

5.1. C# seen in a historic perspective

Lecture 2 - slide 2

It is important to realize that C# stands on the shoulders of other similar object-oriented programming languages. Most notably, C# is heavily inspired by Java. Java, in turn, is inspired by C++, which again - on the object-oriented side - can be traced back to Simula (and, of course, to C on the imperative side).

Here is an overview of the most important object-oriented programming languages from which C# has been derived:

- Simula (1967)
  - The very first object-oriented programming language
- C++ (1983)
  - The first object-oriented programming language in the C family of languages
- Java (1995)
  - Sun's object-oriented programming language
- C# (2001)
  - Microsoft's object-oriented programming language

5.2. The Common Language Infrastructure

Lecture 2 - slide 3

The Common Language Infrastructure (CLI) is a specification that allows several different programming languages to be used together on a given platform. The CLI has a lot of components, typically referred to by three-letter abbreviations (acronyms). Here are the most important parts of the Common Language Infrastructure:

- Common Intermediate language (CIL) including a common type system (CTS)
- Common Language Specification (CLS) - shared by all languages
- Virtual Execution System (VES)
- Metadata about types, dependent libraries, attributes, and more

The following illustration, taken from Wikipedia, illustrates the CLI and its context.

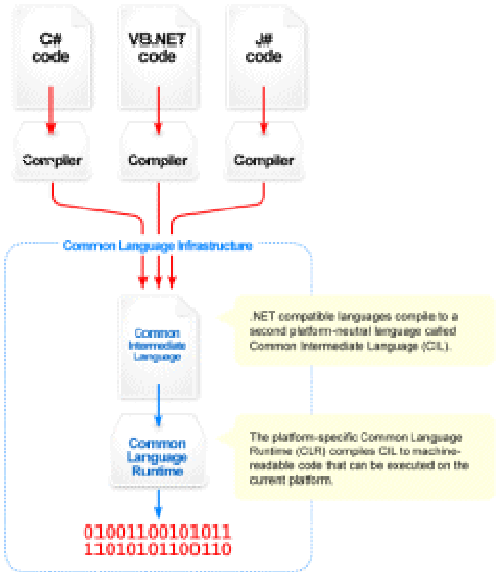

Figure 5.1 *Wikipedia's overview diagram of the CLI*

.Net is one particular implementation of the Common Language Infrastructure, and it is undoubtedly the most complete one. .Net is closely associated with Windows. .Net is, however, not the only implementation of the CLI. Mono is another one, which is intended to work on several platforms. Mono is the primary implementation of the CLI on Linux. Mono is also available on Windows.

MONO and .NET are both implementations of the Common Language Infrastructure

The C# language and the Common Language Infrastructure are standardized by ECMA and ISO

5.3. C# Compilation and Execution

Lecture 2 - slide 5

The Common Language Infrastructure supports a two-step compilation process

- Compilation
  - The C# compiler: Translation of C# source to CIL
  - Produces **`.dll`** and **`.exe`** files
  - *Just in time* compilation: Translation of CIL to machine code
- Execution
  - With interleaved *Just in Time* compilation
  - On Mono: Explicit activation of the interpreter
  - On Window: Transparent activation of the interpreter

`.dll` and **`.exe`** files are - with some limitations - portable in between different platforms

6. C# in relation to C

As already mentioned in Chapter 1, this material is primarily targeted at people who know the C programming language. With this outset, we do not need to dwell on issues such as elementary types, operators, and control structures. The reason is that C and C# (and Java for that sake) are similar in these respects.

In this chapter we will discuss the aspects of C# which have obvious counterparts in C. Hopefully, the chapter will be helpful to C programmers who have an interest in C# programming.

In this chapter 'C#' refers to C# version 2.0. When we discuss C we refer to ANSI C ala 1989.

6.1. Simple types

Lecture 2 - slide 7

C supports the simple types `char`, `bool`, `int`, `float` and `double`. In addition there are a number of variation of some of these. In this context, we will also consider pointers as simple types.

The major differences between C# and C with respect to simple types are the following:

- All simple C# types have fixed bit sizes
- C# has a boolean type called **`bool`**
- C# chars are 16 bit long
- In C# there is a high-precision 128 bit numeric fixed-point type called **`decimal`**
- Pointers are not supported in the normal parts of a C# program
    - In the unsafe part C# allows for pointers like in C
- All simple types are in reality structs in C#, and therefore they have members

In C it is not possible to tell the bit sizes of the simple types. In some C implementations an `int`, for instance, will made by 4 bytes (32 bits), but in other C implementations an `int` may be longer or shorter. In C# (as well as in Java) the bit sizes of the simple types are defined and fixed as part of the specification of the language.

In C there is no boolean type. Boolean false is represented by zero values (such as integer 0) and boolean true is represented by any other integer value (such as the integer 1). In C# there is a boolean type, named `bool`, that contain the natural values denoted by `true` and `false`.

The handling of characters is messy in C. Characters in C are supported by the type named `char`. The `char` type is an integer type. There is a great deal of confusion about signed and unsigned characters. Typically, characters are represented by 8 bits in C, allowing for representation of the extended ASCII alphabet. In C# the type `char` represent 16 bits characters. In many respects, the C# type `char` corresponds to the Unicode alphabet. However, 16 bits are not sufficient for representation of all characters in the Unicode alphabet. The issue of character representation, for instance in text files, relative to the type `char` is a complex issue in C#. In this material it will be discussed in the lecture about IO, starting in Chapter 37. More specifically, you should consult Section 37.7.

The high-precision, 128 bit type called `decimal` is new in C#. It is a decimal floating point type (as opposed to `float` and `double` which are binary floating point types). Values in type `decimal` are intended for financial calculations. Internally, a decimal value consists of a sign bit (representing positive or negative), a 96 bit integer (mantissa) and a scaling factor (exponent) implicitly between 10^0 and 10^{-28}. The 96 bit integer part allows for representation of (at least) 28 decimal digits. The decimal exponent allows you to set the decimal point anywhere in the 28 decimal number. The decimal type uses 3 * 4 = 12 bytes for the mantissa and 4 bytes for the exponent. (Not all bits in the exponent are used, however). For more information, see [decimal-floating-point].

C pointers are not intended to be used in C#. However, C pointers are actually supported in the part of C# known as the unsafe part of the language. The concept of references is very important in C#. References and pointers are similar, but there are several differences as well. Pointers and references will be contrasted and discussed below, in Section 6.5.

All simple types in C# are in reality represented as structs (but not all structs are simple types). As such, this classifies the simple types in C# as *value types*, as a contrast to *reference types*. In addition, in C#, this provides for definition of methods of simple types. Structs are discussed in Section 6.6.

Below we show concrete C# program fragments which demonstrate some aspects of simple types.

```
1  using System;
2
3  class BoolDemo{
4
5    public static void Main(){
6      bool b1, b2;
7      b1 = true; b2 = default(bool);
8      Console.WriteLine("The value of b2 is {0}", b2);  // False
9    }
10
11 }
```

Program 6.1 *Demonstrations of the simple type bool in C#.*

In Program 6.1 we have emphasized the parts that relate to the type `bool`. We declare two boolean variables `b1` and `b2`, and we initialize them in the line below their declarations. Notice the possibility of asking for the default value of type `bool`. This possibility also applies to other types. The output of Program 6.1 reveals that the default value of type `bool` is false.

```
using System;

class CharDemo{

  public static void Main(){
    char ch1 = 'A',
         ch2 = '\u0041',
         ch3 = '\u00c6', ch4 = '\u00d8', ch5 = '\u00c5',
         ch6;

    Console.WriteLine("ch1 is a letter: {0}", char.IsLetter(ch1));

    Console.WriteLine("{0} {1} {2}", ch3, ch4, char.ToLower(ch5));

    ch6 = char.Parse("B");
    Console.WriteLine("{0} {1}", char.GetNumericValue('3'),
                                 char.GetNumericValue('a'));
  }

}
```

Program 6.2 *Demonstrations of the simple type char in C#.*

In Program 6.2 we demonstrate the C# type `char`. We declare a number of variables, `ch1` ... `ch6`, of type `char`. `ch1` ... `ch5` are immediately initialized. Notice the use of single quote character notation, such as `'A'`. This is similar to the notation used in C. Also notice the **`'\u....'`** *escape notation*. This is four digit unicode character notation. Each dot in **`'\u....'`** must be a hexadecimal digit between 0 and f (15). The unicode notation can be used to denote characters, which are not necessarily available on your keyboard, such as the Danish letters Æ, Ø and Å shown in Program 6.2. Notice also the `char` operations, such as `char.IsLetter`, which is applied on `ch1` in the program. Technically, `IsLetter` is a static method in the struct `Char` (see Section 6.6 and Section 14.3 for an introduction to structs). There are many similar operations that classify characters. These operations correspond to the abstractions (macros) in the C library `ctype.h`. It is recommended that you - as an exercise - locate `IsLetter` in the C# library documentation. It is important that you are able to find information about already existing types in the documentation pages. See also Exercise 2.1.

Number Systems and Hexadecimal Numbers **FOCUS BOX 6.1**

In the program that demonstrated the type char we have seen examples of hexadecimal numbers. It is worthwhile to understand why hexadecimal numbers are used for these purposes. This side box is a crash course on number systems and hexadecimal numbers.

The normal numbers are decimal, using base 10. The meaning of the number 123 is

$$1 * 10^2 + 2 * 10^1 + 3 * 10^0$$

The important observation is that we can use an arbitrary base b, $b > 1$ as an alternative to 10 for decomposition of a number. Base numbers which are powers of 2 are particularly useful. If we use base 2 we get the binary numbers. Binary numbers correspond directly to the raw digital representation used in computers. The binary notation of 123 is 1111011 because

$1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

is equal to the decimal number 123.

Binary numbers are important when we approach the lower level of a computer, but as can be seen above, binary numbers are unwieldy and not very practical. Hexadecimal numbers are used instead. Hexadecimal numbers use base 16, which is 2^4. We need 16 digits for notation of hexadecimal numbers. The first 10 digits are 0 .. 9. In lack of better notation, we use the letters A .. F as the six last digits. A = 10, ..., F = 15.

The important observation is that *a group of four binary digits (corresponding to four bits) can be translated to a single hexadecimal number*. Thus, we can immediately translate the binary number 01111011 to the two hexadecimal digits 7 and 11. These two hexadecimal digits are denoted as 7 and B respectively. With this observation, at single byte of eight bits can written as exactly two hexadecimal digits. Grouping the bits of 1111011 leads to 0111 1011. 0111 is 7. 1011 is 11 which is denoted by the hexadecimal digit B. The hexadecimal number 7B means

$7 * 16^1 + 11 * 16^0$

which is 123 (in decimal notation).

The explantion above is - in a nutshell - the reason why you should care about hexadecimal numbers. In Exercise 2.2 we will write programs that deal with hexadecimal numbers.

```
using System;
using System.Globalization;

class NumberDemo{

  public static void Main(){
    sbyte        sb1 = sbyte.MinValue;       // Signed 8 bit integer
    System.SByte sb2 = System.SByte.MaxValue;
    Console.WriteLine("sbyte: {0} : {1}", sb1, sb2);

    byte        b1 = byte.MinValue;          // Unsigned 8 bit integer
    System.Byte b2 = System.Byte.MaxValue;
    Console.WriteLine("byte: {0} : {1}", b1, b2);

    short        s1 = short.MinValue;        // Signed 16 bit integer
    System.Int16 s2 = System.Int16.MaxValue;
    Console.WriteLine("short: {0} : {1}", s1, s2);

    ushort         us1 = ushort.MinValue;    // Unsigned 16 bit integer
    System.UInt16  us2= System.UInt16.MaxValue;
    Console.WriteLine("ushort: {0} : {1}", us1, us2);

    int          i1 = int.MinValue;          // Signed 32 bit integer
    System.Int32 i2 = System.Int32.MaxValue;
    Console.WriteLine("int: {0} : {1}", i1, i2);

    uint           ui1 = uint.MinValue;      // Unsigned 32 bit integer
    System.UInt32  ui2= System.UInt32.MaxValue;
    Console.WriteLine("uint: {0} : {1}", ui1, ui2);

    long         l1 = long.MinValue;         // Signed 64 bit integer
    System.Int64 l2 = System.Int64.MaxValue;
    Console.WriteLine("long: {0} : {1}", l1, l2);

    ulong           ul1 = ulong.MinValue;    // Unsigned 64 bit integer
```

```
36      System.UInt64   ul2= System.UInt64.MaxValue;
37      Console.WriteLine("ulong: {0} : {1}", ul1, ul2);
38
39      float           f1 = float.MinValue;     // 32 bit floating-point
40      System.Single   f2= System.Single.MaxValue;
41      Console.WriteLine("float: {0} : {1}", f1, f2);
42
43      double          d1 = double.MinValue;    // 64 bit floating-point
44      System.Double   d2= System.Double.MaxValue;
45      Console.WriteLine("double: {0} : {1}", d1, d2);
46
47      decimal         dm1 = decimal.MinValue;  // 128 bit fixed-point
48      System.Decimal  dm2= System.Decimal.MaxValue;
49      Console.WriteLine("decimal: {0} : {1}", dm1, dm2);
50
51
52      string s = sb1.ToString(),
53             t = 123.ToString();
54
55    }
56
57
58  }
```

Program 6.3 *Demonstrations of numeric types in C#.*

In Program 6.3 we show a program that demonstrates all numeric types in C#. For illustrative purposes, we use both the simple type names (such as `int`, shown in **purple**) and the underlying struct type names (such as `System.Int32` shown in **blue**). To give you a feeling of the ranges of the types, the program prints the smallest and the largest value for each numeric type. At the bottom of Program 6.3 we show how the operation `ToString` can be used for conversion from a numeric type to the type `string`. The output of the numeric demo program is shown in Listing 6.4 (only on web).

Hexadecimal Numbers in C# **FOCUS BOX 6.2**

In Focus box 6.1 we studied hexadecimal numbers. We will now see how to deal with hexadecimal numbers in C#.

A number prefixed with 0x is written in hexadecimal notation. Thus, 0x123 is equal to the decimal number 291.

In C and Java the prefix 0 is used for octal notation. Thus, in C and Java 0123 is equal to the decimal number 83. This convention is not used in C#. In C#, 0123 is just a decimal number prefixed with a redundant digit 0.

While *prefixes* are used for encoding of number systems, *suffixes* of number constants are used for encoding of numerical types. As an example, 0X123L denotes a hexadecimal constant of type `long` (a 64 bit integer). The following suffixes can be used for integer types: **U** (unsigned), **L** (long), and **UL** (unsigned long). The following suffixes can be used for real types: **F** (float), **D** (double), and **M** (decimal). Both lowercase and uppercase suffixes will work.

A number can formatted in both decimal and hexadecimal notation. In the context of a `Console.WriteLine` call, the format specification (or placeholder) `{i:X}` will write the value of the variable `i` in hexadecimal

notation. This is demonstrated by the following C# program:

```
using System;
class NumberDemo{
  public static void Main(){
     int i = 0123,
         j = 291;
     long k = 0X123L;

     Console.WriteLine("{0:X}", i);   // 7B
     Console.WriteLine("{0:D}", i);   // 123
     Console.WriteLine("{0:X}", j);   // 123
     Console.WriteLine("{0:D}", k);   // 291
  }
}
```

In the program shown above, **D** means decimal and **X** means hexadecimal. Some additional formattings are also provided for numbers: **C** (currency notation), **E** (exponential notation), **F** (fixed point notation), **G** (Compact general notation), **N** (number notation), **P** (percent notation), **R** (round trip notation for float and double). You should consult the online documentation for additional explanations.

Exercise 2.1. *Exploring the type Char*

The type `System.Char` (a struct) contains a number of useful methods, and a couple of constants.

Locate the type `System.Char` in your C# documentation and take a look at the methods available on characters.

You may ask where you find the C# documentation. There are several possibilities. You can find it at the Microsoft MSDN web site at `msdn.microsoft.com`. It is also integrated in Visual Studio and - to some degree - in Visual C# express. It comes with the C# SDK, as a separate browser. It is also part of the documentation web pages that comes with Mono. If you are a Windows user I will recommend the Windows SDK Documentation Browser which is bundled with the C# SDK.

Along the line of the character demo program above, write a small C# program that uses the `char` predicates `IsDigit`, `IsPunctuation`, and `IsSeparator`.

It may be useful to find the code position - also known as the *code point* - of a character. As an example, the code position of 'A' is 65. Is there a method in `System.Char` which gives access to this information? If not, can you find another way to find the code position of a character?

Be sure to understand the semantics (meaning) of the method `GetNumericValue` in type `Char`.

Exercise 2.2. *Hexadecimal numbers*

In this exercise we will write a program that can convert between decimal and hexadecimal notation of numbers. Please consult the focus boxes about hexadecimal numbers in the text book version if you need to.

You might expect that this functionality is already present in the C# libraries. And to some degree, it is.

The static method `ToInt32(string, Int32)` in class `Convert` converts the string representation of a number (the first parameter) to an arbitrary number system (the second parameter). Similar methods exist for other integer types.

The method `ToString(string)` in the struct `Int32`, can be used for conversion from an integer to a hexadecimal number, represented as a string. The parameter of `ToString` is a format string. If you pass the string "X" you get a hexadecimal number.

The program below shows examples:

```
using System;
class NumberDemo{
  public static void Main(){
     int i = Convert.ToInt32("7B", 16);     // hexadecimal 7B (in base 16) ->
                                            // decimal 123
     Console.WriteLine(i);                  // 123

     Console.WriteLine(123.ToString("X"));  // decimal 123 -> hexadecimal 7B
  }
}
```

Now, write a method which converts a list (or array) of digits in base 16 (or more generally, base *b*, *b* >= 2) to a decimal number.

The other way around, write a method which converts a positive decimal integer to a list (or array) of digits in base 16 (or more generally, base *b*).

Here is an example where the requested methods are used:

```
  public static void Main(){
    int r = BaseBToDecimal(16, new List{7, 11});  // 7B  -> 123
    List s = DecimalToBaseB(16, 123);             // 123 -> {7, 11} = 7B
    List t = DecimalToBaseB(2, 123);              // 123 -> {1, 1, 1, 1, 0, 1, 1 } =
                                                       // 1111011
    Console.WriteLine(r);
    foreach (int digit in s) Console.Write("{0} ", digit);  Console.WriteLine();
    foreach (int digit in t) Console.Write("{0} ", digit);
  }
```

6.2. Enumerations types

Lecture 2 - slide 8

Enumeration types in C# are similar to enumeration types in C, but a number of extensions have been introduced in C#:

- Enumeration types of several different *underlying types* can be defined (not just `int`)
- Enumeration types inherit a number of methods from the type **`System.Enum`**
- The symbolic enumeration constants can be printed (not just the underlying number)
- Values, for which no enumeration constant exist, can be dealt with
- Combined enumerations represent a collection of enumerations

Below, in Program 6.5 we see that the enumeration type `OnOff` is based on the type `byte`. The enumeration type `Ranking` is - per default - based on `int`.

```
1  using System;
2
3  class NonSimpleTypeDemo{
4
5    public enum Ranking {Bad, OK, Good}
6
7    public enum OnOff: byte{
8      On = 1, Off = 0}
9
10   public static void Main(){
11     OnOff status = OnOff.On;
12     Console.WriteLine();
13     Console.WriteLine("Status is {0}", status);
14
15     Ranking r = Ranking.OK;
16     Console.WriteLine("Ranking is {0}", r  );
17     Console.WriteLine("Ranking is {0}", r+1);
18     Console.WriteLine("Ranking is {0}", r+2);
19
20     bool res1 = Enum.IsDefined(typeof(Ranking), 3);
21     Console.WriteLine("{0} defined: {1}", 3, res1);
22
23     bool res2= Enum.IsDefined(typeof(Ranking), Ranking.Good);
24     Console.WriteLine("{0} defined: {1}", Ranking.Good , res2);
25
26     bool res3= Enum.IsDefined(typeof(Ranking), 2);
27     Console.WriteLine("{0} defined: {1}", 2 , res3);
28
29     foreach(string s in Enum.GetNames(typeof(Ranking)))
30        Console.WriteLine(s);
31   }
32
33 }
```

Program 6.5 *Demonstration of enumeration types in C#.*

In the example the methods `IsDefined` and `GetNames` are examples of static methods inherited from `System.Enum`.

In line 13 of Program 6.5 `On` is printed. In a similar C program, the number 1 would be printed.

In line 16 OK is printed, and line 17 prints Good. In line 18 the value of `r + 2` is 3, which does not correspond to any of the values in type `Ranking`. Therefore the base value 3 is printed.

All the output of Program 6.5 is listed in Listing 6.6.

Combined enumeration (sometimes known as *flags enumeration*) is a slightly more advanced concept. We introduce it in Focus box 6.3.

Let us point out some additional interesting details in Program 6.5. There are two enumeration types in the program, namely a type called `Ranking` and another called `OnOff`. When we declare variables, the types `Ranking` and `OnOff` are used via their names. C programmer will be relieved to find out that it is not necessary (and not possible) to write `enum Ranking` and `enum OnOff`. Thus, no C-like `typedefs` are necessary to introduce simple naming.

In order to disambiguate the referencing of constants in an enumeration type, dot notation ala `Ranking.OK` must always be used. In the same way as in C, the enumeration constants have associated an integer value. The operation `IsDefined` allows us to check if a given value belongs to an enumeration type. `IsDefined` is an operation (a method) in a struct called `Enum`.

As a very pleasant surprise for the C programmer, there is access to the names of enumeration constants from the program. We show in the program that the expressions `Enum.GetNames(typeof(Ranking))` returns a string array with the elements "Bad", "OK", and "Good". In the same direction - as we have already seen above - it is possible to print the symbolic names of the enumeration constants. This is very useful. In C programs we always need a tedious `switch` to obtain a similar effect..

Combined Enumerations **FOCUS BOX 6.3**

Combined enumerations can be used to deal with small sets symbolic constants. Here is an example:

```
[Flags]
public enum Direction: byte{
  North = 1, East = 2, South = 4, West = 8,
}
```

The first thing to notice is the mapping of symbolic constants to powers of two. We can form an expression `North | West` which is the bitwise combination of the underlying integer values 1 and 8. `|` is a bitwise or operator, see Table 6.1. At the binary level, 1 | 8 is equivalent to 0001 | 1000 = 1001, which represents the number 9. You should think of 1001 as a bit array where the leftmost bit is the encoding of `West` and the rightmost bit is the encoding of `North`.

`[Flags]` is an application of an attribute, see Section 39.6. It instructs the compiler to generate symbolic names of combinations, such as the composite name `North, West` in the example below.

We can program with the enumeration type in the following way:

```
Direction d = Direction.North | Direction.West;
Console.WriteLine("Direction {0}", d);                // Direction North, West
Console.WriteLine("Direction {0}", (int)d);           // 9
Console.WriteLine(HasDirection(d, Direction.North));  // True
Console.WriteLine(HasDirection(d, Direction.South));  // False
```

The method `HasDirection` is a method we have programmed in the following way:

```
// Is d in the direction e
public static bool HasDirection(Direction d, Direction e){
  return (d & e) == e;
}
```

It checks if `e` is contained in `d` in the a bitwise sense. It is also possible to name some of the combinations explicitly in the enumeration type:

```
[Flags]
public enum Direction: byte{
  North = 1, East = 2, South = 4, West = 8,
  NorthWest = North | West, NorthEast = North | East,
  SouthWest = South | West, SouthEast = South | East
}
```

Exercise 2.3. *ECTS Grades*

Define an enumeration type `ECTSGrade` of the grades A, B, C, D, E, Fx and F and associate the Danish 7-step grades 12, 10, 7, 4, 2, 0, and -3 to the symbolic ECTS grades.

What is the most natural *underlying type* of `ECTSGrade`?

Write a small program which illustrates how to use the new enumeration type.

Exercise 2.4. *Use of Enumeration types*

Consult the documentation of type type `System.Enum`, and get a general overview of the methods in this struct.

Be sure that you are able to find the documentation of `System.Enum`

Test drive the example `EnumTest`, which is part of MicroSoft's documentation. Be sure to understand the program relative to its output.

Write your own program with a simple enumeration type. Use the `Enum.CompareTo` method to compare two of the values of your enumeration type.

6.3. Non-simple types

Lecture 2 - slide 9

The most important non-simple types are defined by classes and structs. These types define non-atomic objects/values. Because C# is a very rich language, there are other non-simple types as well, such as interfaces and delegates. We will not discuss these in this chapter, but they will play important roles in later chapters.

The most important similarities between C and C# with respect to non-simple types can be summarized in the following way:

- Arrays in C#: Indexed from 0. Jagged arrays - arrays of arrays
- Strings in C#: Same notation as in C, and similar escape characters
- Structs in C#: A value type like in C.

The most important differences are:

- Arrays: Rectangular arrays in C#
- Strings: No `\0` termination in C#
- Structs: Much expanded in C#. Structs can have methods.

A C programmer, who have experience with arrays, strings, and structs from C, will immediately feel comfortable with these types in C#. But such a C programmer will also quickly find out, that there are substantial new possibilities in C# that makes life easier.

Arrays and strings will be discussed in Section 6.4. Classes and structs are, basically, what the rest of the book is about. The story about classes starts in Chapter 10.

6.4. Arrays and Strings

Lecture 2 - slide 10

Arrays and strings are both non-simple types that are well-known by C programmers. In C# both arrays and strings are defined by classes. As we will see later, this implies that arrays and strings are represented as objects, and they are accessed via references. This stands as a contrast to C# structs, which are values and therefore not accessed via references.

The syntax of array declaration and initialization is similar in C and C#. In C, a string is a pointer to a the first character in the string, and it is declared of the type `char*`. C# supports a type named `string`. The notation of string constants is also similar in C and C#, although C# offers additional possibilities (the `@"..."` notation, see below).

The following summarizes important differences in between C and C# with respect to arrays and strings:

- Arrays in C# can be rectangular or jagged (arrays of arrays)
- In C#, an array is not a pointer to the first element
- Index out of bound checking is done in C#
- Strings are immutable in C#, but not in C
- In C# there are two kinds of string literals: `"a string\n"` and `@"a string\n"`

A multi-dimensional array in C is constructed as an array in which the elements are themselves arrays. Such arrays are known as jagged arrays in C#, because not all constituent arrays need to have the same size. In

addition C# supports a new kind of arrays, namely rectangular arrays (of two or more dimensions). Such arrays are similar to arrays in Pascal.

C is inherently unsafe, in part because indexes out of bounds are not systematically caught at run-time. C# is safe in this respect. An index out of bound in a running C# program raises an exception, see Chapter 36.

C programmers may be puzzled by the fact that strings are immutable in C#. Once a string is constructed, it is not possible to modify the character elements of the string. This is also a characteristic of strings in Java. This makes it possible to share a given string in several contexts. The bookkeeping behind this is called *interning*. (You can, for instance, read about interning in the documentation of the static method `String.Intern`). In case mutable strings are necessary, the class `System.Text.StringBuilder` makes them available.

The well-known double quote string notation is used both in C and C#. Escape characters, prefixed with backslashes (such as in `"\n"`) are used in C as well and in C#. C# supports an alternative notation, called *verbatim string constants*, `@"..."` , in which the only escape notation is `""` which stands for the double quote character itself. Inside a verbatim string constant it possible to have newline characters, and all backslashes appear as backslash characters in the string. An example of a verbatim strings will be shown in Program 6.9.

Below, in Program 6.7 we will demonstrate a number of aspects of arrays in C#.

```
using System;

class ArrayStringDemo{

  public static void Main(){
     string[]  a1,
               a2 = {"a", "bb", "ccc"};

     a1 = new string[]{"ccc", "bb", "a"};

     int[,]    b1 = new int[2,4],
               b2 = {{1,2,3,4}, {5,6,7,8}};

     double[][] c1 = { new double[]{1.1, 2.2},
                       new double[]{3.3, 4.4, 5.5, 6.6} };

     Console.WriteLine("Array lengths. a1:{0} b2:{1} c1:{2}",
                        a1.Length, b2.Length, c1.Length);

     Array.Clear(a2,0,3);

     Array.Resize<string>(ref a2,10);
     Console.WriteLine("Lenght of a2: {0}", a2.Length);

     Console.WriteLine("Sorting a1:");
     Array.Sort(a1);
     foreach(string str in a1) Console.WriteLine(str);
  }

}
```

Program 6.7 *Demonstrations of array types in C#.*

We declare two variables, `a1` and `a2`, of the type `string[]`. In other words, `a1` and `a2` are both arrays of strings. `a1` is not initialized in its declaration. (Local variables in C# are not initialized to any default value). `a2` is initialized by means of an *array initializer*, namely `{"a", "bb", "ccc"}`. The length of the array is determined by the number of expressions within the pair of curly braces.

The arrays `b1` and `b2` are both rectangular 2 times 4 arrays.

The array `c1` is an example of a jagged array. `c1[1]` is an array of length 2. `c1[2]` is an array of length 4.

Next we try out the `Length` operation on `a1`, `b2` and `c1`. The result is a1:3 `b2:8 c1:2`. Please notice and understand the outcome.

Finally we demonstrate a number of additional operations on arrays: `Clear`, `Resize`, and `Sort`. These are all methods in the class `System.Array`.

The output of the array demo program is shown in Listing 6.8 (only on web).

Arrays, as discussed above, will be used in many of your future programs. But as an alternative, you should be aware of the collection classes, in particular the type parameterized, "generic" collection classes. These classes will be discussed in Chapter 45.

Now we will study a program example that illustrates uses of the type `string`.

```
1  using System;
2
3  class ArrayStringDemo{
4
5    public static void Main(){
6       string s1        = "OOP";
7       System.String s2 = "\u004f\u004f\u0050";   // equivalent
8       Console.WriteLine("s1 and s2: {0} {1}", s1, s2);
9
10      string s3 = @"OOP on
11             the \n semester ""Dat1/Inf1/SW3""";
12      Console.WriteLine("\n{0}", s3);
13
14      string s4 = "OOP on \n            the \\n semester \"Dat1/Inf1/SW3\"";
15      Console.WriteLine("\n{0}", s4);
16
17      string s5 = "OOP E06".Substring(0,3);
18      Console.WriteLine("The substring is: {0}", s5);
19   }
20
21 }
```

Program 6.9 *A demonstration of strings in C#.*

The strings `s1` and `s2` in Program 6.9 contain the same three characters, namely 'O', 'O', and 'P'.

Similarly, the strings referred by `s3` and `s4` are equal to each other (in the sense that they contain the same sequences of characters). As already mentioned above, the string constant in line 10-11 is a *verbatim string constant*, in which an escape sequence like `\n` denotes itself. In verbatim strings, only "" has a special interpretation, namely as a single quoute character.

Finally, in Program 6.9, the `Substring` operation from the class `System.String` is demonstrated.

The output of the string demonstration program in Program 6.9 is shown in Listing 6.10 (only on web).

Exercise 2.5. *Use of array types*

Based on the inspiration from the accompanying example, you are in this exercise supposed to experiment with some simple C# arrays.

First, consult the documentation of the class `System.Array`. Please notice the properties and methods that are available on arrays in C#.

Declare, initialize, and print an array of names (e.g. array of strings) of all members of your group.

Sort the array, and search for a given name using `System.Array.BinarySearch` method.

Reverse the array, and make sure that the reversing works.

Exercise 2.6. *Use of string types*

Based on the inspiration from the accompanying example, you are in this exercise supposed to experiment with some simple C# strings.

First, consult the documentation of the class `System.String` - either in your documentation browser or at `msdn.microsoft.com`. Read the introduction (remarks) to string which contains useful information! There exists a large variety of operations on strings. Please make sure that you are aware of these. Many of them will help you a lot in the future!

Make a string of your own first name, written with escaped Unicode characters (like we did for "OOP" in the accompanying example). If necessary, consult the unicode code charts (Basic Latin and Latin-1) to find the appropriate characters.

Take a look at the `System.String.Insert` method. Use this method to insert your last name in the first name string. Make your own observations about `Insert` relative to the fact that strings in C# are immutable.

6.5. Pointers and references

Lecture 2 - slide 11

Pointers are instrumental in almost all real-life C programs, both for handling dynamic memory allocation, and for dealing with arrays. Recall that an array in C is simply a pointer to the first element of the array.

References in C# (and Java) can be understood as a restricted form of pointers. C# references are never explicitly dereferenced, references are not coupled to arrays, and references cannot be operated on via the arithmetic C# operators; There are no pointer arithmetic in (the safe part of) C#. As a special notice to C++ programmers: References in C# have nothing to do with references in C++.

Here follows an itemized overview of pointers and references.

- Pointers
  - In normal C# programs: Pointers are not used
    - All the complexity of pointers, pointer arithmetic, dereferencing, and the address operator is not found in normal C# programs
  - In specially marked unsafe sections: Pointers can be used almost as in C.
    - Do not use them in your C# programs!
- References
  - Objects (instance of classes) are always accessed via references in C#
  - References are automatically dereferenced in C#
  - There are no particular operators in C# that are related to references

Program 6.11 shows some basic uses of references in C#. The variables `cRef` and `anotherCRef` are declared of type `C`. `C` happens to be an almost trivial class that we have defined in line 3-5. Classes are reference types in `C` (see Chapter 13). `cref` declared in in line 11 is assigned to `null` (a reference to nothing) in line 12. Next, in line 15, `cref` is assigned to a new instance of `C`. Via the reference in `cref` we can access the members `x` and `y` in the `C` object, see line 18. We can also pass a reference as a parameter to a function `F` as in line 19. This does not copy the referenced object when entering `F`.

```
1  using System;
2
3  public class C {
4    public double x, y;
5  }
6
7  public class ReferenceDemo {
8
9    public static void Main(){
10
11     C cRef, anotherCRef;
12     cRef = null;
13     Console.WriteLine("Is cRef null: {0}", cRef == null);
14
15     cRef = new C();
16     Console.WriteLine("Is cRef null: {0}", cRef == null);
17
18     Console.WriteLine("x and y are ({0},{1})", cRef.x, cRef.y);
19     anotherCRef = F(cRef);
20   }
21
22   public static C F(C p){
23     Console.WriteLine("x and y are ({0},{1})", p.x, p.y);
24     return p;
25   }
26
27 }
```

Program 6.11 *Demonstrations of references in C#.*

The output of Program 6.11 is shown in Listing 6.12 (only on web).

There is no particular complexity in normal C# programs due to use of references

6.6. Structs

Lecture 2 - slide 12

Structs are well-known by C programmers. It is noteworthy that arrays and structs are handled in very different ways in C. In C, arrays are deeply connected to pointers. Related to the discussion in this material, we will say that pointers are dealt with by *reference semantics*, see Section 13.1. Structs in C are dealt with by *value semantics*, see Section 14.1. Structs are copied by assignment, parameter passing, and returns. Arrays are not!

Let us now compare structs in C and C#:

- Similarities
  - Structs in C# can be used almost in the same way as structs in C
  - Structs are *value types* in both C and C#
- Differences
  - Structs in C# are almost as powerful as classes
    - Structs in C# can have operations (methods) in the same way as classes
    - Structs in C# cannot inherit from other structs or classes

In Program 6.13 we see a program with a struct called `Point`. The variable `p1` contains a point (3.0, 4.0). Because structs are value types, `p1` does not refer to a point. It *contains* the two coordinates of type `double` that represents the points. `p2` is uninitialized. In line 15, `p1` is copied into `p2`. This is a field-by-field (bit-by-bit) copy operation. No manipulation of references is involved. Finally we show the activation of a method `Mirror` on `p2`. Hereby the state of the second point is mutated to (-3,-4).

```
1  using System;
2
3  public struct Point {
4    public double x, y;
5    public Point(double x, double y){this.x = x; this.y = y;}
6    public void Mirror(){x = -x; y = -y;}
7  } // end Point
8
9  public class StructDemo{
10
11  public static void Main(){
12    Point p1 = new Point(3.0, 4.0),
13          p2;
14
15    p2 = p1;
16
17    p2.Mirror();
18    Console.WriteLine("Point is: ({0},{1})", p2.x, p2.y);
19  }
20 }
```

Program 6.13 *Demonstrations of structs in C#.*

6.7. Operators

Lecture 2 - slide 13

Expressions in C# have much in common with expressions in C. Non-trivial expressions are built with use of operators. We summarize the operators in C# in Table 6.1. As it appears, there is a substantial overlap with the well-known operators i C. Below the table we will comment on the details.

Level	Category	Operators	Associativity (binary/tertiary)
14	*Primary*	`x.y  f(x)  a[x]  x++  x--` `new  typeof  checked  unchecked  default  delegate`	*left to right*
13	*Unary*	`+  -  !  ~  ++x  --x  (T)x  true  false  sizeof`	*left to right*
12	*Multiplicative*	`*  /  %`	*left to right*
11	*Additive*	`+  -`	*left to right*
10	*Shift*	`<<  >>`	*left to right*
9	*Relational and Type testing*	`<  <=  >  >=  is  as`	*left to right*
8	*Equality*	`==  !=`	*left to right*
7	*Logical/bitwise and*	`&`	*left to right*
6	*Logical/bitwise xor*	`^`	*left to right*
5	*Logical/bitwise or*	`|`	*left to right*
4	*Conditional and*	`&&`	*left to right*
3	*Conditional or*	`||`	*left to right*
2	*Null coalescing*	`??`	*left to right*
1	*Conditional*	`?:`	*right to left*
0	*Assignment or Lambda expression*	`=  *=  /=  %=  +=  -` `=  <<=  >>=  &=  ^=  |=  =>`	*right to left*

Table 6.1 *The operator priority table of C#. Operators with high level numbers have high priorities. In a given expression, operators of high priority are evaluated before operators with lower priority. The associativity tells if operators at the same level are evaluated from left to right or from right to left.*

The operators shown in red are new and specific to C#. The operator `new` creates an instance (an object) of a class or it initializes a value of struct. We have already encountered `new` in some of the simple demo programs, for instance Program 6.11 and Program 6.13. See Section 12.2 for details on `new`. The operators `is`, `as` and `typeof` will not be discussed here. Please refer to Section 28.12 for details on these. The operations `checked` and `uncheked` relate to the safe and unsafe part of C# respectively. In this material we only deal with the safe part, and therefore these two C# operators can be disregarded. The `default` operator gives access to the default value of value types, see Section 12.3. The `delegate` operator is used for definition of anonymous functions, see Section 22.1. The unary `true` and `false` operators tell when a value in a user

defined type is regarded as *true* or *false* respectively. See Section 21.2 for more details. The expression `x ?? y` is a convenient shortcut of `x != null ? x : y`. See Section 14.9 for more details on **??**. **=>** is the operator which is used to form lambda expressions in C#3.0, see Section 22.4.

A couple of C operators are not part of C#. The address operator `&` and the dereference operator `*` are not found in (the safe part of) C# (but they are actually available in the unsafe part of the language). They are both related to pointers, and as discussed in Section 6.5 pointers are not supported in (the safe part of) C#.

All the remaining operators should be familiar to the C programmer.

The operators listed above have fixed and predefined meanings when used together with primitive types in C#. On top of these it is possible to define new meanings of some of the operators on objects/values of your own types. This is called *operator overloading*, and it is discussed in more details in Chapter 21. The subset of overloadable operators is highlighted in Table 21.1.

6.8. Commands and Control Structures

Lecture 2 - slide 14

Almost all control structures in C can be used the same way in C#

Commands (also known as `statements`) are able to mutate the program state at run-time. As such, the most important command is the assignment. The commands constitute the "imperative heart" of the programming language. The control structures provide means for sequencing the commands.

The commands and control structures of C# form - to a large extent - a superset of the commands and control structures of C. Thus, the knowledge of commands and control structures in C can be used directly when learning C#.

As usual, we will summarize similarities and differences between C and C#. The similarities are the following:

- Expression statements, such as **`a = a + 5;`**
- Blocks, such as **`{a = 5; b = a;}`**
- **`if`**, **`if-else`**, **`switch`**, **`for`**, **`while`**, **`do-while`**, **`return`**, **`break`**, **`continue`**, and **`goto`** in C# are all similar to C

As in C, an expression becomes a command if it is followed by a semicolon. Therefore we have emphasized the semicolon above in the assignment `a = a + 5;`

As it will be clear from Program 6.15 below, the **switch** control structures in C and C# differ substantially.

The main differences between C and C# regarding control structures are the following:

- The C# **`foreach`** loop provides for easy traversal of all elements in a collection
- **`try-catch-finally`** and **`throw`** in C# are related to exception handling

- Some more specialized statements have been added: **`checked`**, **`unchecked`**, **`using`**, **`lock`** and **`yield`**.

The **foreach** control structures is an easy-to-use version of a for loop, intended for start-to-end traversal of collections. We will not here touch on **try-catch-finally** and **throw**. Please refer to our coverage of exception handling in Section 36.2 for a discussion of these.

Let us now look at some concrete program examples with control structures. In the examples below, program fragments shown in **red** color illustrate erroneous aspects. Program fragments shown in **green** are all right.

```
/* Right, Wrong */
using System;

class IfDemo {

  public static void Main(){
    int i = 0;

    /*
    if (i){
      Console.WriteLine("i is regarded as true");
    }
    else {
      Console.WriteLine("i is regarded as false");
    }
    */

    if (i != 0){
      Console.WriteLine("i is not 0");
    }
    else {
      Console.WriteLine("i is 0");
    }
  }
}
```

Program 6.14 *Demonstrations of if.*

The **if-else** control structure has survived from C. Program 6.14 in reality illustrates a difference between handling of boolean values of C and C#. This has already been treated in Section 6.1. The point is that an expression of non-`bool` type (such the integer `i`) cannot be used as the controlling expression of an **if-else** control structure i C#.

Let us now look at a program with **switch** control structures. As already mentioned earlier there are a number of noteworthy differences between C and C# regarding **switch**.

```
/* Right, Wrong */
using System;

class SwitchDemo {
  public static void Main(){
    int j = 1, k = 1;

    /*
    switch (j) {
      case 0:  Console.WriteLine("j is 0");
      case 1:  Console.WriteLine("j is 1");
```

```
12        case 2:  Console.WriteLine("j is 2");
13        default: Console.WriteLine("j is not 0, 1 or 2");
14      }
15      */
16
17      switch (k) {
18        case 0:  Console.WriteLine("m is 0"); break;
19        case 1:  Console.WriteLine("m is 1"); break;
20        case 2:  Console.WriteLine("m is 2"); break;
21        default: Console.WriteLine("m is not 0, 1 or 2"); break;
22      }
23
24      switch (k) {
25        case 0: case 1:  Console.WriteLine("n is 0 or 1"); break;
26        case 2: case 3:  Console.WriteLine("n is 2 or 3"); break;
27        case 4: case 5:  Console.WriteLine("n is 4 or 5"); break;
28        default: Console.WriteLine("n is not 1, 2, 3, 4, or 5"); break;
29      }
30
31      string str = "two";
32      switch (str) {
33        case "zero":  Console.WriteLine("str is 0"); break;
34        case "one":  Console.WriteLine("str is 1"); break;
35        case "two":  Console.WriteLine("str is 2"); break;
36        default: Console.WriteLine("str is not 0, 1 or 2"); break;
37      }
38    }
39  }
```

Program 6.15 *Demonstrations of switch.*

The first switch in Program 6.15 is legal in C, but it is illegal i C#. It illustrates the *fall through problem*. If `j` is 0, case 0, 1, 2, and `default` are all executed in a C program. Most likely, the programmer intended to write the second switch, starting in line 17, in which each `case` is broken with use of the `break` command. In C# the compiler checks that each branch of a switch never encounters the ending of the branch (and, thus, never falls through to the succeeding branch).

The third switch in the demo program shows that two or more cases can be programmed together. Thus, like in C, it is legal to fall trough empty cases.

The final switch shows that it is possible to switch on strings in C#. This is very convenient in many contexts! In C, the type of the switch expression must be integral (which means an integer, char, or an enumeration type).

Let us also mention that C# allows special goto constructs (**`goto case`** *constant* and **`goto default`**) inside a switch. With use of these it is possible to jump from one case to another, and it is even possible to program a loop inside a switch (by jumping from one case to an already executed case). It is recommended only to use these specialized goto constructs in exceptional situations, or for programming of particular patterns (in which it is natural to organize a solution around multiple branches that can pass the control to each other).

Next we will study a program that illustrates the **foreach** loop.

```
1  /* Right, Wrong */
2  using System;
3
4  class ForeachDemo {
5    public static void Main(){
6
```

```
7       int[] ia = {1, 2, 3, 4, 5};
8       int sum = 0;
9
10      foreach(int i in ia)
11        sum += i;
12
13      Console.WriteLine(sum);
14    }
15  }
```

Program 6.16 *Demonstrations of foreach.*

As mentioned above, **`foreach`** is a variant of a for loop that traverses all elements of a collection. (See how this is provided for in Section 31.6). In the example of Program 6.16 all elements of the array are traversed. Thus, the loop variable `i` will be 1, 2, 3, 4 and 5 in succession. Many efforts in C# have been directed towards supporting foreach on the collection types that you program yourself. Also notice that loop control variable, `i`, is declared inside the foreach construct. This cannot be done in a conventional for loop in C (although it can be done in C99 and in C++).

Finally, we will see an example of **try-catch**.

```
1  /* Right, Wrong */
2  using System;
3
4  class TryCatchDemo {
5    public static void Main(){
6      int i = 5, r = 0, j = 0;
7
8      /*
9      r = i / 0;
10     Console.WriteLine("r is {0}", r);
11     */
12
13     try {
14       r = i / j;
15       Console.WriteLine("r is {0}", r);
16     } catch(DivideByZeroException e){
17         Console.WriteLine("r could not be computed");
18     }
19   }
20 }
```

Program 6.17 *Demonstrations of try catch.*

Division by 0 is a well-known cause of a run-time error. Some compilers are, in some situations, even smart enough to identify the error at compile-time. In Program 6.17 the erroneous program fragment never reaches the activation of `WriteLine` in line 10, because the division by zero halts the program.

The expression `i / j`, where `j` is 0, is embedded in a **try-catch** control structure. The division by zero raises an exception in the running program, which may be handled in the catch part. The `WriteLine` in line 17 is encountered in this part of the example. Thus, the program survives the division by zero. Later in the material, starting in Chapter 33, we discuss - in great details - errors and error handling and the use of **try-catch**.

Before we leave the assignments and control structure we want to mention the *definite assignment* rule in C#. The rule states that every declared variable must be assigned to a value before the variable is used. The compiler enforces the rule. Take a look at the program below.

```
1  using System;
2
3  class DefiniteAssignmentDemo{
4
5    public static void Main(){
6      int a, b;
7      bool c;
8
9      if (ReadFromConsole("Some Number") < 10){
10        a = 1; b = 2;
11     } else {
12        a = 2;
13     }
14
15     Console.WriteLine(a);
16     Console.WriteLine(b);  // Use of unassigned local variable 'b'
17
18     while (a < b){
19       c = (a > b);
20       a = Math.Max(a, b);
21     }
22
23
24     Console.WriteLine(c);  // Use of unassigned local variable 'c'
25
26   }
27
28   public static int ReadFromConsole(string prompt){
29     Console.WriteLine(prompt);
30     return int.Parse(Console.ReadLine());
31   }
32 }
```

Program 6.18 *Demonstrations of definite assignment.*

The program declares three variables `a`, `b`, and `c` in line 6-7, without initializing them. Variable `b` is used in line 16, but it cannot be guarantied that the **`if-else`** control structure in line 9-13 assigns a value to the variable `b`. Therefore, using a *conservative approach*, the compiler complains about line 16. The error message is emphasized in the comment at the end of line 16.

Similarly, the variable `c` declared in line 7 is not necessarily assigned by the **`while`** control structure in line 18-21. Recall that if `a >= b` when we enter the while loop, the line 19 and 20 are never executed. This explains the error message associated to line 24.

The definite assignment rule, enforced by the compiler, implies that we never get run-time errors due to uninitialized variables. On the other hand, the rule also prevents some program from executing on selected input. If the number read in line 9 of Program 6.18 is less than 10 both `b` and `c` will be assigned when used in the `WriteLine` calls.

6.9. Functions

Lecture 2 - slide 15

Functions are the primary abstractions in C. In C# "function" (or "function member") is the common name of a variety of different kinds of abstractions. The most well-known of these is known as methods. The others are properties, events, indexers, operators, and constructors.

Functions in C# belong to types: classes or structs. Thus, functions in C# cannot be freestanding like in C. Functions are always found inside a type.

The conceptual similarities between functions in C and methods in C# are many and fundamental. In our context it is, however, most interesting to concentrate on the differences:

- Several different parameter passing techniques in C#
  - Call by value. For input. No modifier.
  - Call by reference. For *input and output* or *output only*
    - Input and output: Modifier **`ref`**
    - Output: Modifier **`out`**
    - Modifiers used both with formal and actual parameters
- Functions with a variable number of input parameters in C# (cleaner than in C)
- Overloaded function members in C#
- First class functions (delegates) in C#

In C all parameters are passed by value. However, passing a pointer by value in C is often proclaimed as *call by reference*. In C# there are several parameter passing modes: *call by value* and two variants of *call by reference* (`ref` and `out` parameters). The default parameter passing mode is call by value. *Call by reference* parameter passing in C (via pointers) is not the same as `ref` parameters in C#. `ref` parameters in C# are much more like Pascal `var` (variable) parameters.

In C it is possible, but messy, to deal with functions of a variable (or an arbitrary) number of arguments. In C# this is easier and cleaner. It is supported by the `params` keyword in a formal parameter list. An example is provided in Program 6.20.

A function in C is identified by its name. A method in C# is identified by its name together with the types of the formal parameters (the so-called *method signature*). This allows several methods with the same names to coexist, provided that their formal parameter types differ. A set of equally named methods (with different formal parameter types) is known as *overloaded* methods.

A function in C# can be handled without naming it at all. Such functions are known as delegates. Delegates come from the functional programming language paradigm, where functions most often are *first class objects*. Something of *first class* can be passed as parameters, returned as results from functions, and organized in data structures independent of naming. Delegates seem to be more and more important in the development of C#. In C# 3.0 the nearby concepts of *lambda expressions* and *expression trees* have emerged. We have much more to say about delegates later in this material, see Chapter 22.

```
1  /* Right, Wrong */
2
3  using System;
4
5
6  /*
7  public int Increment(int i){
8    return i + 1;
9  }
10
11 public void Main (){
12   int i = 5,
13       j = Increment(i);
14   Console.WriteLine("i and j: {0}, {1}", i, j);
15 } // end Main
```

```
*/

public class FunctionDemo {

  public static void Main (){
    SimpleFunction();
  }

  public static void SimpleFunction(){
    int i = 5,
        j = Increment(i);
    Console.WriteLine("i and j: {0}, {1}", i, j);
  }

  public static int Increment(int i){
    return i + 1;
  }
}
```

Program 6.19 *Demonstration of simple functions in C#.*

Program 6.19 shows elementary examples of functions (methods) in a C# program. The program text decorated with **red** color shows two functions, `Main` and `Increment`, outside of any type. This is illegal in C#.

Shown in **green** we again see the function `Increment`, now located in a legal context, namely inside the type (class) `FunctionDemo`. The function `SimpleFunction` calls `Increment` in a straightforward way. The function `Main` serves as *main program* in C#. It is here the program starts. We see that `Main` calls `SimpleFunction`.

```
using System;
public class FunctionDemo {

  public static void Main (){
    ParameterPassing();
  }

  public static void ValueFunction(double d){
    d++;}

  public static void RefFunction(ref double d){
    d++;}

  public static void OutFunction(out double d){
    d = 8.0;}

  public static void ParamsFunction(out double res,
                                    params double[] input){
    res = 0;
    foreach(double d in input) res += d;
  }

  public static void ParameterPassing(){
    double myVar1 = 5.0;
    ValueFunction(myVar1);
    Console.WriteLine("myVar1: {0:f}", myVar1);            // 5.00

    double myVar2 = 6.0;
    RefFunction(ref myVar2);
    Console.WriteLine("myVar2: {0:f}", myVar2);            // 7.00

    double myVar3;
    OutFunction(out myVar3);
```

```
34      Console.WriteLine("myVar3: {0:f}", myVar3);            // 8.00
35
36      double myVar4;
37      ParamsFunction(out myVar4, 1.1, 2.2, 3.3, 4.4, 5.5);  // 16.50
38      Console.WriteLine("Sum in myVar4: {0:f}", myVar4);
39    }
40
41  }
```

Program 6.20 *Demonstration of parameter passing in C#.*

The four functions in Program 6.20, `ValueFunction`, `RefFunction`, `OutFunction`, and `ParamsFunction` demonstrate the different parameter passing techniques of C#.

The call-by-value parameter `d` in `ValueFunction` has the same status as a local variable in `ValueFunction`. Therefore, the call of `ValueFunction` with `myVar1` as actual parameter does not affect the value of `myVar1`. It does, however, affect the value of `d` in `ValueFunction`, but this has no lasting effect outside `ValueFunction`. In a nutshell, this is the idea of call by value parameters.

In `RefFunction`, the formal parameter `d`, is a **ref** parameter. The corresponding actual parameter must be a variable. And indeed it is a variable in our sample activation of `RefFunction`, namely the variable named `myVar2`. Inside `RefFunction`, the formal parameter `d` is an *alias* of the actual parameter (`myVar2`). Thus, the incrementing of `d` actually increments `myVar2`. Pascal programmers will be familiar with this mechanism (via var parameters) but C programmers have not encountered this before - at least not while programming in C.

`OutFunction` demonstrates the use of an **out** parameter. **out** parameters are similar to **ref** parameters, but only intended for data output from the function. More details of **ref** and **out** parameters appears in Section 20.6 and Section 20.7.

Notice that in C#, the keywords **ref** and **out** must be used both in the formal parameter list and in the actual parameter list. This is nice, because you will hereby spot the parameter passing mode in calls. In most other programming language it is necessary to consult the function definition to find out about the parameter passing modes of the parameters involved.

The first parameter of `ParamsFunction`, `res`, is an **out** parameter, intended for passing the sum of the `input` parameter back to the caller. The formal **param** parameter, `input`, must be an array. The similar actual parameters (occurring at the end of the actual parameter list) are inserted as elements into a new array, and bound to the formal parameter `input`. With this mechanism, an arbitrary number of "rest parameters" (of the same or comparable types) can be handled, and bundled into an array in the C# function, which is being called.

Program 6.21 (only on web) shows a class with four methods, all of which are named `F`. These functions are distinguished by different formal parameters, and by different parameter passing modes. Passing an integer value parameter activates the first `F`. Passing a double value parameter activates the second `F`. Passing a double and a bool (both as values) activates the third `F`. Finally, passing an integer **ref** parameter activates the fourth `F`.

6.10. Input and output

Lecture 2 - slide 16

In C, the functions `printf` and `scanf` are important for handling output to the screen, input from the keyboard, and file IO as well. It is therefore interesting for C programmers to find out how the similar facilities work in C#.

In C#, the `Console` class encapsulates the streams known as *standard input* and *standard output*, which per default are connected to the keyboard and the screen. The various write functions in the `Console` class are quite similar to the `printf` function in C. The `Console` class' read functions are not as advanced as `scanf` in C. There is not direct counterpart of the C `scanf` function in C#.

First, in Program 6.22 we will study uses of the `Write` and `WriteLine` functions.

```
1  /* Right, Wrong */
2
3  using System;
4  public class OutputDemo {
5
6  // Placeholder syntax: {<argument#>[,<width>][:<format>[<precision>]]}
7
8    public static void Main(){
9      Console.Write(    "Argument number only: {0} {1}\n", 1, 1.1);
10 //  Console.WriteLine("Formatting code d: {0:d},{1:d}", 2, 2.2);
11
12     Console.WriteLine("Formatting codes d and f: {0:d}  {1:f}", 3, 3.3);
13     Console.WriteLine("Field width: {0,10:d}  {1,10:f}", 4, 4.4);
14     Console.WriteLine("Left aligned: {0,-10:d}  {1,-10:f}", 5, 5.5);
15     Console.WriteLine("Precision: {0,10:d5}  {1,10:f5}", 6, 6.6);
16     Console.WriteLine("Exponential: {0,10:e5}  {1,10:e5}", 7, 7.7);
17     Console.WriteLine("Currency: {0,10:c2}  {1,10:c2}", 8, 8.887);
18     Console.WriteLine("General: {0:g}  {1:g}", 9, 9.9);
19     Console.WriteLine("Hexadecimal: {0:x5}", 12);
20
21     Console.WriteLine("DateTime formatting with F: {0:F}", DateTime.Now);
22     Console.WriteLine("DateTime formatting with G: {0:G}", DateTime.Now);
23     Console.WriteLine("DateTime formatting with T: {0:T}", DateTime.Now);
24   }
25 }
```

Program 6.22 *Demonstrations of Console output in C#.*

Like `printf` in C, the methods `Write` and `WriteLine` accept a control string and a number of additional parameters which are formatted and inserted into the control string. `Write` and `WriteLine` actually rely on an underlying `Format` method in class `String`. Notice that a there exists many overloaded `Write` and `WriteLine` methods in the class `Console`. Here we concentrate of those that take a string - the control string - as the first parameter.

The following call of `printf` in C

```
printf("x: %d, y: %5.2f, z: %Le\n", x, y, z);
```

is roughly equivalent to the following call of of `WriteLine` in C#

```
Console.WriteLine("x: {0:d}, y: {1,5:F2}, z: {2:E}", x, y, z);
```

The equivalence assumes that `x` is of type int, `y` is a float, and that `z` is a long double.

The general syntax of a *placeholder* (the stuff in between pairs of curly braces) in a C# formatting string is

{<argument#>[,<width>][:<format>[<precision>]]}

where `[...]` denotes optionality (zero or one occurrence).

C programmers do often experience strange and erroneous formatting of output if the conversion characters (such as `d`, `f`, and `e` in the example above) are inconsistent with the actual type of the corresponding variables or expressions (`x`, `y`, and `z` in the example). In C#, such problems are caught by the compiler, and as such they do not lead to wrong results. This is a much needed improvement.

Let us briefly explain the examples in line 9-23 of Program 6.22. In line 9 the default formatting is used. This corresponds to the letter code `g`. In line 10 an error occurs because the code `d` only accepts integers. The number 2.2 is not an integer. In line 13 we illustrate use of width 10 for an integer and a floating-point number. Line 14 is similar, but it uses left justification (because the width is negative). Line 15 illustrates use of the precision 5 for an integer and a floating-point number. In line 16 we format two numbers in exponential (scientific) notation. In line 17 we illustrate formatting of currencies (kroner or dollars, for instance, dependent on the culture setting). Line 18 corresponds to line 9. Line 19 calls for hexadecimal formatting of a number.

One way to learn more about output formatting is to consult the documentation of the static method `Format` in class `System.String`. From there, goto Formatting Overview. Later in this material, in Section 31.7, we will see how we can program custom formatting of our own types.

The last three example lines in Program 6.22 illustrate formatting of objects of type `DateTime` in C#. Such objects represent at point in time. In the example, the expression `DateTime.Now` denotes the current point in time.

The output of Program 6.22 is shown in Listing 6.23 (only on web).

We now switch from output to input.

```
1  /* Right, Wrong */
2
3  using System;
4  public class InputDemo {
5
6    public static void Main(){
7      Console.Write("Input a single character: ");
8      char ch = (char)Console.Read();
9      Console.WriteLine("Character read: {0}", ch);
10     Console.ReadLine();
11
12     Console.Write("Input an integer: ");
13     int i  = int.Parse(Console.ReadLine());
14     Console.WriteLine("Integer read: {0}", i);
15
16     Console.Write("Input a double: ");
17     double d  = double.Parse(Console.ReadLine());
18     Console.WriteLine("Double read: {0:f}", d);
19   }
20 }
```

Program 6.24 *Demonstrations of Console input in C#.*

In Program 6.24 `Console.Read()` reads a single character. The result returned is a positive integer, or -1 if no character can be read (typically because we are located at the end of an input file). `Read` blocks until **enter** is typed. Non-blocking input is also available via the method `Console.ReadKey`. The expression `Console.ReadLine()` reads a line of text into a string. The last two, highlighted examples show how to read a text string and, via the `Parse` method in type `int` and `double`, to convert the strings read to values of type `int` and `double` respectively. Notice that `scanf` in C can take hand of such cases.

The output of Program 6.24 is shown in Listing 6.25 (only on web).

Later in this material we have much more to say about input and output in C#. See Chapter 37 - Chapter 39 . The most important concept, which we will deal with in these chapters, is the various kinds of streams in C#.

6.11. Comments

Lecture 2 - slide 17

We finalize our comparison of C and C# with an overview of the different kinds of C# comments. Recall that C only supports delimited comments (although C programmers also often use single-line comments, which actually is used in C++ and in newer versions of C (C99)).

C# supports two different kinds of comments and XML variants of these:

- **Single-line comments like in C++**
  ```
  // This is a single-line comment
  ```
- **Delimited comments like in C**
  ```
  /* This is a delimited comment */
  ```
- **XML single-line comments:**
  ```
  /// <summary> This is a single-line XML comment </summary>
  ```
- **XML delimited comments:**
  ```
  /** <summary> This is a delimited XML comment </summary> */
  ```

XML comments can only be given before declarations, not inside other fragments. XML comments are used for documentation of types. We have much more to say about XML comments in our discussion of documentation of C# programs. Delimited C# comments cannot be nested.

6.12. References

[Decimal-floating-point] Decimal Floating Point in .NET
http://www.yoda.arachsys.com/csharp/decimal.html

7. C# in relation to Java

C# is heavily inspired by Java. In this section we will, at an overall level, compare Java and C#. The goal of this relatively short chapter is to inform Java programmers about similarities and differences in relation to C#. It is recommended that you already have familiarized yourself with C# in relation to C, as covered in Chapter 6.

In this chapter 'Java' refers to version to 5.0 and 'C#' refers to C# version 2.0.

7.1. Types

Lecture 2 - slide 20

In Java, types can be defined by classes and interfaces. This is also possible in C#. In addition, C# makes it possible to define structs and delegates to which there are no counterparts in Java. Java supports sophisticated, class based enumeration types. Enumeration types in C# are simpler, and relative close to enumeration types in C. - This is the short story. After the itemized summary, we will compare the two languages more carefully.

The similarities and differences with respect to types can be summarized in this way:

- Similarities.
  - Classes in both C# and Java
  - Interfaces in both C# and Java
- Differences.
  - Structs in C#, not in Java
  - Delegates in C#, not in Java
  - Nullable types in C#, not in Java
  - Class-like Enumeration types in Java; Simpler approach in C#

If you have written a Java program with classes and interfaces, it is in most cases relatively easy to translate the program to C#. In this material we discuss classes in C# in Chapter 11 an interfaces in Chapter 31

There are no structs in Java. Structs in C# are, at the outset, similar to structs in C. (See Section 6.6 for a discussion of C structs versus C# structs). Your knowledge of C structs is a good starting point for working with structs in C#. However, structs in C# are heavily extended compared with C. As an important observation, C# structs have a lot in common with classes. Most important, there are operations (methods) and constructors in both C# classes and C# structs. It is also possible to control the visibility of data and operations in both classes and structs. Structs in C# are value types, in the meaning that instances of structs are contained in variables, and copied by assignments and in parameter passings. In contrast, classes are reference types, in the meaning that instances of classes are accessed by references. Instances of classes are not copied in assignments and in parameter passings. For more details on structs see Chapter 14, in particular Section 14.3.

Delegates in C# represents a type of methods. A delegate object can contain a method. More correctly, a delegate can contain a reference to a method. It can actually contain several such references. With use of delegates it becomes possible to treat methods as *data*, in the same way as instance of classes represent data. We can store a method in a variable of delegate type. We can also pass a method as a parameter to another

method. In Java it is not possible pass a method `M` as a parameter to another method. If we need to pass `M`, we have to pass an object of class `C` in which `M` is a method. Needless to say, this is a complicated and contrived way of using function parameters. - In C#, delegates are the foundation of events (see Chapter 23), which, in turn, are instrumental to programming of graphical user interfaces in C# and certain design patters, not least the ***Observer*** (see Section 24.1). For more details on delegate types see Chapter 22.

Nullable types relate to value types, such as structs. A variable of a struct type `S` cannot contain the value `null`. In contrast, a variable of class type `C` can contain the value `null`. The nullable `S` type, denoted `S?`, is a variant of `S` which includes the `null` value. For more details, see Section 14.9.

Enumeration types in both C# and Java allow us to associate symbolic constants to values in an integer type. We demonstrated enumeration types in C# in Section 6.2 of the previous chapter. In Java, an enumeration type is a special form of a class. Each enumerated value is an instance of this special class. Consequently, an enumeration type in Java is a reference type. In C# an enumeration type is a value type. - As a Java-historic remark, enumeration types did not exist in early versions Java. Enumeration types were simulated by a set of `final static` variables (one `final static` variable for each value in the enumeration type). The support of enumeration types shifted dramatically in Java 1.5: from almost no support in previous versions to heavy support via special classes. It is remarkable that the Java designers have chosen to use so many efforts on enumeration types!

7.2. Operations

Lecture 2 - slide 21

Operations in Java programs are defined by methods that belong to classes. This is our only possibility of defining operations in Java. In C# there exists several additional possibilities. In this material we have devoted an entire lecture 'Data Access and Operations' (from Chapter 17 to Chapter 24) to these issues.

The similarities and differences with respect to operations can be summarized in this way:

- Similarities: Operations in both Java and C#
  - Methods
- Differences: Operations only in C#
  - Properties
  - Indexers
  - Overloaded operators
  - Anonymous methods

As already mentioned above, C# methods can be defined in both classes and structs. It is not possible to define local methods in methods - neither in C# or Java. The closest possibility in C# is use of anonymous methods, see below.

Properties provide for getters and setters of fields (instance variables as well as class variables - static variables) in C#. In Java it is necessary to define methods for getting and setting of instance variables. A single property in C# can either be a getter, a setter, or both. From an application point of view, properties are used as though we access of the variables of a class/object directly (as it would be possible if the variables were public). For more details on properties see Chapter 18.

Indexers can be understood as a special kind of properties that define array-like access to objects and values in C#. The notation `a[i]` and `a[i] = x` is well-know when the name `a` denotes an array and if `i` is an integer. In C# we generalize this notation to arbitrary instances of classes and structs denoted by `a`. With an indexer we program the meaning of accessing the i'th element of `a` (`a[i]`) and the meaning of setting the i'th element of `a` (`a[i] = ...`). Indexers are discussed in Chapter 19.

In most languages, the operators like `+`, `-`, `<`, and `&` have fixed meanings, and they only work with the simple types (such as `int`, `bool`, `char`, etc). We reviewed the C# operators in Section 6.7, and as it appears, operators in C, Java, and C# have much in common. In Java, the operators only work for certain predefined types, and you cannot change the meaning of these operators. In C# it is possible to use the existing operator symbols for operations in your own types. (You cannot invent new operator symbols, and you cannot change the precedence nor the associativity of the symbols). We say that the operators in C# can be *overloaded*. For instance, in C# it would be possible to define the meaning of `aBankAccount + aTriangle`, where `aBankAccout` refers to an instance of class `BankAccount` and `aTriangle` refers to an instance of class `GeometricShape`. When the existing operator symbols are natural in relation to our own classes, the idea of overloaded operators is great. In other situations, overloaded operators do not add much value.

We are used to a situation where procedures, functions, and methods have names. In both Java and C# we can define named methods in classes. In C#, we can also define named methods in structs. In C# it is possible to define non-named methods as well. As part of arbitrary expressions, we can create a function or method. Such a function or method is called a delegate. As indicated by the name, delegates are closely related to delegate types, as discussed above in Section 7.1. For more details on this topic see Chapter 22 and in particular the program examples of Section 22.1.

7.3. Other substantial differences

Lecture 2 - slide 22

In addition to the overall differences in the area of types and operations, as discussed in the two previous sections, there are a number of other substantial differences between Java and C#. The most important of these differences are summarized below.

- Program organization
  - No requirements to source file organization in C#
- Exceptions
  - No *catch or specify* requirement in C#
- Nested and local classes
  - Classes inside classes are static in C#: No *inner classes* like in Java
- Arrays
  - Rectangular arrays in C#
- Virtual methods
  - Virtual as well as non-virtual methods in C#

In Java there is a close connection between classes and sources files. It is usually recommended that there is only one class per file, but the rule is actually that there can be one public and several non-public classes per Java source file. The proper name of the source file should be identical to the name of the public class. Likewise, there is a close connection between packages and directories. A package consists of the classes whose source files belong to a given directory. - The organization of C# programs is different. In C# there is no connection between the names of classes and the name of a C# source files. A source file can contain

several classes. Instead of packages, C# organizes types in namespaces and assemblies. Namespaces are tangible, as they are represented syntactically in the source files. A namespace contains types and/or recursively other (nested) namespaces. Assemblies are produced by the C# compiler. Assemblies represent a 'packaging' mechanism, and assemblies are almost orthogonal to both the file/directory organization and the namespace organization. As it appears, C# uses a much more complex - and presumably more powerful - program organization than Java. We discuss organization of C# programs in Chapter 15 at the end of the lecture about Classes.

Java supports both *checked exceptions* and *unchecked exceptions*, but it is clearly the ideal of Java to work with checked exceptions. (Unchecked exception is also known as RuntimeExceptions). It is natural to ask about the difference. A checked exception must either be handled in the method `M` in which is occurs, or the method `M` must declare that an activation of `M` can cause an exception (of a given type) which callers of `M` need to care about. This is sometimes called the *catch or specify principle*. The most visible consequence of this principle is that Java methods, which do not handle exceptions, must declare that they **throws** specific types of exceptions. Thus, the signature of Java methods include information about the kind errors they may cause. - C# does not adhere to the *catch or specify principle*. In C# all exceptions correspond to the so-called RuntimeExceptions in Java. Exceptions in C# are discussed in Chapter 36.

Java is stronger than C# with respect to class nesting. Both Java and C# support *static nested classes* (using Java terminology). In this setup, the innermost class can only refer to static members of the outer class. In contrast to C#, Java also supports inner *classes*. An instance of an inner class has a reference the instance of the outer class. Inner classes have implications to the object structure. Static nested classes have no such implications. In addition, Java supports *local classes* that are local to a method, and *anonymous classes* which are instantiated on the fly. C# does not.

In both Java and C# it is possible to work with arrays in which the elements themselves are arrays, and so on recursively. Using C# terminology, this is called *jagged arrays*, because it facilitates multi-dimensional arrays of irregular shapes. In contrast to Java, C# in addition supports *rectangular arrays*, in which all rows are of equal lengths. We have already discussed jagged and rectangular arrays in our comparison with C in Section 6.4.

Virtual methods relate to redefinition of methods in class hierarchies (inheritance). In Java all methods are virtual. What this means (for C#) is explained in details in Section 28.14. In C# it is possible to have both virtual and non-virtual methods. This complicates the understanding of inheritance quite a bit. There are, however, good reasons to support both. In Section 32.9 we will review a prominent example where the uniform use of virtual methods in Java runs into trouble.

8. C# in relation to Visual Basic

This chapter is intended for students who have a background in imperative Visual Basic programming. The goal of this chapter is to make the transfer from Visual Basic to C# as easy as possible. We do that by showing and discussing a number of equivalent Visual Basic and C# programs. In this chapter Visual Basic programs are shown on a blue background, and C# programs are shown on a green background. The discussion of equivalent Visual Basic and C# program is textually organized in between the two programs.

In this chapter 'Visual Basic' refers to the version of Visual Basic supported by the .Net Framework version 2.0.

In this edition the comparison of Visual Basic and C# is only available in the web version of the material.

9. C# Tools and IDEs

Many potential C# programmers will be curious about tools and environments (IDEs) for C# programming. Therefore we will, briefly, enumerate the most obvious possibilities. We will mention possibilities in both Windows and Unix.

9.1. C# Tools on Windows

Lecture 2 - slide 41

Windows is the primary platform of C#. This is due to the fact that C# is a Microsoft language.

Microsoft supplies several different set of tools that support the C# programmer:

- .NET Framework SDK 3.5
  - "Software Development Kit"
  - Command line tools, such as the C# compiler **`csc`**
- Visual C# Express
  - IDE - An Integrated Development Environment
  - A C# specialized, *free* version of Microsoft Visual Studio 2008
- Visual Studio
  - IDE - An Integrated Development Environment
  - The professional, *commercial* development environment for C# and other programming languages

The .Net Standard Development Kit (SDK) supports the raw tools, including a traditional C# compiler. Although many programmers today use contemporary IDEs such as Visual Studio or Eclipse, I find it important that all programmers understand the basic and underlying activation of the compiler.

The Visual C# Express edition is a free (gratis) variant of Visual Studio, explicitly targeted at students and other newcomers to C#. There are video resources [cs-video-resources] available for getting started with C# 2008 Express. The experience you get with Visual C# Express can immediately be transferred to Visual studio. The two IDEs are very similar.

Visual Studio is the commercial flagship environment of C# programming. You will have to buy Visual Studio if you want to use it. Notice, however, that many universities have an academic alliance with Microsoft that provides access to Visual Studio and other Microsoft software.

9.2. C# Tools on Unix

Lecture 2 - slide 42

The MONO project provides tools for C# development on Linux, Solaris, Mac OS X, Unix in general, and interesting enough also on Windows. MONO is the choice if you swear to the Linux platform.

Let us summarize the MONO resources, available to the Linux people:

- MONO
  - An *open source* project (sponsored by Novell)
  - Corresponds the the Microsoft SDK
  - Based on ECMA specifications of C# and the Common Language Infrastructure (CLI) Virtual Machine
  - Command line tools
  - Compilers: **`mcs`** (C# 1.5) and **`gmcs`** (C# 2.0)
- MONO on cs.aau.dk
  - Mono is already installed on the application servers at cs.aau.dk
- MONO on your own Linux machine
  - You can install MONO yourself if you wish
- MonoDevelop
  - A GNOME IDE for C#

For good reasons, the MONO CLI is not as updated as the .NET solutions. MONO will most probably always be at least one step behind Microsoft.

9.3. References

[Cs-video-resources] C# Express Video Lectures
http://msdn.microsoft.com/en-us/beginner/bb964631.aspx