Programming Principals

Semester 1, 2022

Dr Alan Woodley

Tarang Janawalkar

Contents

1 Programming

Definition 1.0.1. Programming is the process of designing and building an executable computer program to accomplish a specific computing result or to perform a specific task.

Programming involves:

1. Analysis
2. Design
3. Implementation
4. Testing

1.1 Analysis

- What is the problem?
- What data is involved — input, output?
- What is the relationship between input and output?
- What other constraints?

1.2 Design

- Specify modules that need to be created to implement the solution.
- Module — group of closely related functions and data they need to do their job
- Which parts of the problem are closely related? They probably belong together in a module.
- How do modules fit together and communicate?
- How can I test each of these modules to be sure they behave as desired?
- How can I test the complete system to be sure it behaves as desired?

1.3 Implementation

- Create working software to ”do” each part of the design
- Select suitable algorithms and data structures to do each required item of functionality
- Write code to implement the algorithms and data structures

1.4 Testing

- Before we write any code we should have a very clear idea how the program can be validated; usually that is done by testing

2 Types and Expressions

2.1 Expressions

Definition 2.1.1 (Expressions)**.** An expression is a combination of values, variables and operators. In interactive mode, an interpreter evaluates expressions and displays the result. However, in a script, we must first compile the program to an executable in order to perform any tasks.

Definition 2.1.2 (Type)**.** The type of an expression is "what kind of data" the expression carries.

Definition 2.1.3 (Variables)**.** Variables are a kind of expression which have an **identity** and a **value**.

The **value** of a variable may change as a program runs, however in a statically typed language, the **type** of each variable is specified before it can be used, and never changes.

Variables can be declared as follows

```
TYPE_SPECIFIER IDENTIFIER;
TYPE_SPECIFIER IDENTIFIER = EXPRESSION;
```

In the first instance, we declare the type of the variable without initialising it. In the second case we declare and initialise the variable.

Definition 2.1.4 (Literal)**.** The term *literal* refers to the literal representation of a value. For example, when disambiguating between the variable `dog` and the string `"dog"` we would say the *"variable dog"* vs. the *"string literal dog"*.

C# identifiers must take the following into account

- Identifiers can contain letters, digits and the underscore character (`_`)
- Identifiers must begin with a letter
- Identifiers cannot contain whitespaces
- Identifiers are case sensitive ("`Foo`" and "`foo`" are different variables)
- Reserved words such as C# keywords cannot be used as identifiers

2.2 Types

There are 9 integer and 3 floating-point types in C#, each with a different size and range. The minimum and maximum values of any type can be determined using `TYPE.MinValue` and `TYPE.MaxValue`.

C# type	Size	Range
sbyte	8 bit	-2^7 to $2^7 - 1$
byte	8 bit	0 to $2^8 - 1$
short	16 bit	-2^{15} to $2^{15} - 1$
ushort	16 bit	0 to $2^{16} - 1$
int	32 bit	-2^{31} to $2^{31} - 1$
uint	32 bit	0 to $2^{32} - 1$
long	64 bit	-2^{63} to $2^{63} - 1$
ulong	64 bit	0 to $2^{64} - 1$

Table 1: Integer types in C#.

C# type	Size	Range	Precision
float	32 bit	$\pm1.5 \times 10^{-45}$ to $\pm3.4 \times 10^{38}$	6 to 9 digits
double	64 bit	$\pm5.0 \times 10^{-324}$ to $\pm1.7 \times 10^{308}$	15 to 17 digits
decimal	128 bit	$\pm1.0 \times 10^{-28}$ to 7.9228×10^{28}	28 to 29 digits

Table 2: Floating-point types in C#.

2.3 Type Conversion

By default, C# automatically assigns the **int**, **uint**, **long**, or **ulong** type to any integer depending the size and sign of the provided number. Any floating-point number is instantiated as a **double**.

```
$ (100).GetType()
System.Int32
$ (4294967295).GetType()
System.UInt32
$ (-4294967295).GetType()
System.Int64
$ (100.0).GetType()
System.Double
```

To override this behaviour we can add a suffix to the number.

Type	Suffix
uint	u
long	l
ulong	u, l or ul
float	f
double	d
decimal	m

Table 3: Type suffixes for numeric types.

If a literal is prefixed with `u`, its type is the first of the following types in which its value can be represented: **`uint`**, **`ulong`**.

Similarly, if a literal is prefixed with `l`, its type is the first of the following types in which its value can be represented: **`long`**, **`ulong`**.

If the value of an integer is within the range of the destination type, the value can be implicitly converted to the remaining integer types.

2.3.1 Implicit Conversion

Implicit conversions do not require any special syntax as the conversion always succeeds and no data is lost. The following diagram illustrates implicit conversions for numeric types. The direction of the arrows indicate possible implicit conversions where intermediate types can be skipped. Note that all integer types can be converted to floating-point types.

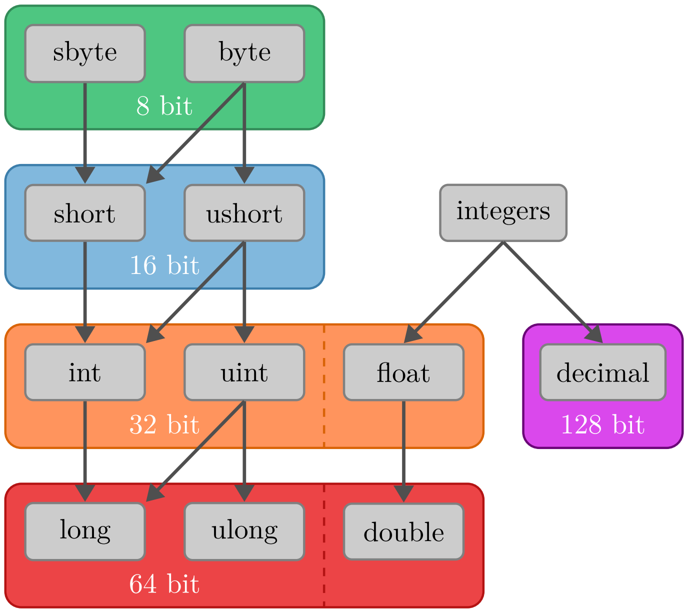

Figure 1: Numeric type implicit conversions in C#.

For example

```
$ // 8 bit unsigned integer to 64 bit signed integer
$ byte b = 32; Console.WriteLine($"{b} -- {b.GetType()}")
32 -- System.Byte
$ long l = b; Console.WriteLine($"{l} -- {l.GetType()}")
32 -- System.Int64

$ // 16 bit signed integer to double precision floating-point number
$ short s = 30000; Console.WriteLine($"{s} -- {s.GetType()}")
30000 -- System.Int16
```

```
$ double d = s; Console.WriteLine($"{d} -- {d.GetType()}")
30000 -- System.Double
```

2.3.2 Explicit Conversion

When a conversion cannot be made without risking losing information, the compiler requires that we perform an explicit conversion using a **type cast**. The syntax for a type cast is as follows

```
(NEW_TYPE) EXPRESSION
```

For example

```
$ // Decimal to single precision floating-point number
$ decimal pi = 3.14159265358979323m; Console.WriteLine($"{pi} -- {pi.GetType()}")
3.14159265358979323 -- System.Decimal
$ float fPi = (float) pi; Console.WriteLine($"{fPi} -- {fPi.GetType()}")
3.141593 -- System.Single

$ // 32 bit unsigned integer to 8 bit signed integer
$ uint u = 9876; Console.WriteLine($"{u} -- {u.GetType()}")
9876 -- System.UInt32
$ byte b = (byte) u; Console.WriteLine($"{b} -- {b.GetType()}")
148 -- System.Byte
```

In the final example, to understand what is happening in the explicit conversion, we must look at the binary representation of the two integers.

```
$ uint u = 9876; Console.WriteLine(Convert.ToString(u, 2).PadLeft(32, '0'))
00000000000000000010011010010100
$ byte b = (byte) u; Console.WriteLine(Convert.ToString(b, 2).PadLeft(8, '0'))
10010100
```

Notice that the value is determined by copying the 8 least significant bits from the 32 bit unsigned integer.

2.4 Operators

The following table lists the C# operators starting with the highest precedence to the lowest.

Operators	**Category**
`x.y`, `f(x)`, `a[i]`, `x++`, `x--`, `x!`, `x->y` and other keywords	Primary
`+x`, `-x`, `!x`, `~x`, `++x`, `--x`, `^x`, `(T)x`, **`await`**, `&x`, `*x`, **`true`**, **`false`**	Unary
`x..y`	Range
`switch`, **`with`**	—
`x * y`, `x / y`, `x % y`	Multiplicative
`x + y`, `x - y`	Additive
`x << y`, `x >> y`	Shift
`x < y`, `x > y`, `x <= y`, `x >= y`, **`is`**, **`as`**	Relational and type-testing
`x == y`, `x != y`	Equality
`x & y`	Logical `AND`
`x ^ y`	Logical `XOR`
`x | y`	Logical `OR`
`x && y`	Conditional `AND`
`x || y`	Conditional `OR`
`x ?? y`	Null-coalescing operator
`c ? t : f`	Conditional operator
`x = y`, `=>` and shorthand assignments	Assignment and lambda declaration

Table 4: Precedence of various operators in C#.

In C#, arithmetic operations behave as expected.

```
$ 123 + 12
135 // System.Int32
$ 123 - 12
111 // System.Int32
$ 123 * 12
1476 // System.Int32
$ 123 / 12
10 // System.Int32
$ 123 % 12
3 // System.Int32
```

Binary operators always convert the resulting data type to the data type of the argument with the largest size in memory (with a few exceptions when converting between floating-point types). Hence division between two integers truncates any floating-point precision.

```
$ 123 / 12
10 // System.Int32
$ 123.0 / 12
10.25 // System.Double
$ 123 / 12.0
10.25 // System.Double
$ 123.0 / 12.0
```

```
10.25 // System.Double
```

Caution should be used when dividing two numbers to avoid loss of precision.