SIT232 - Object Oriented Development

SESSION 3. METHODS AND ARRAYS

Outline

Session 03. Methods and arrays

- Objectives
- Collections
- Method overloading
- Recursion
- Argument promotion and casting
- Validity checking

Session 3. Methods and arrays

Objectives

At the end of this session you should:

- Be familiar **collections** and be able to ***apply them including arrays, generic collections, the foreach loop***, and command line arguments;
- Understand ***static and constant members*** as being ***associated with a class rather than objects*** and be able to apply them in your programs;
- Be able to apply ***pass-by-value, pass-by-reference, method overloading, and recursion*** in your programs;
- Understand the ***concept of scope and how it applies to names in C#*** programs;
- Understand how ***argument promotion and casting are used for data type conversion;*** and
- Be able to introduce basic ***validity checking*** into your programs

Collections

Collections are regularly needed when developing an application

- An order ***contains many*** books
- An investment account ***contains many*** stocks
- An class ***has many*** students
- An flight ***has many*** passengers

We consider two:

- Arrays: available in *most programming languages*
- List<>: generic collection ***provided by Microsoft.Net***

Collections

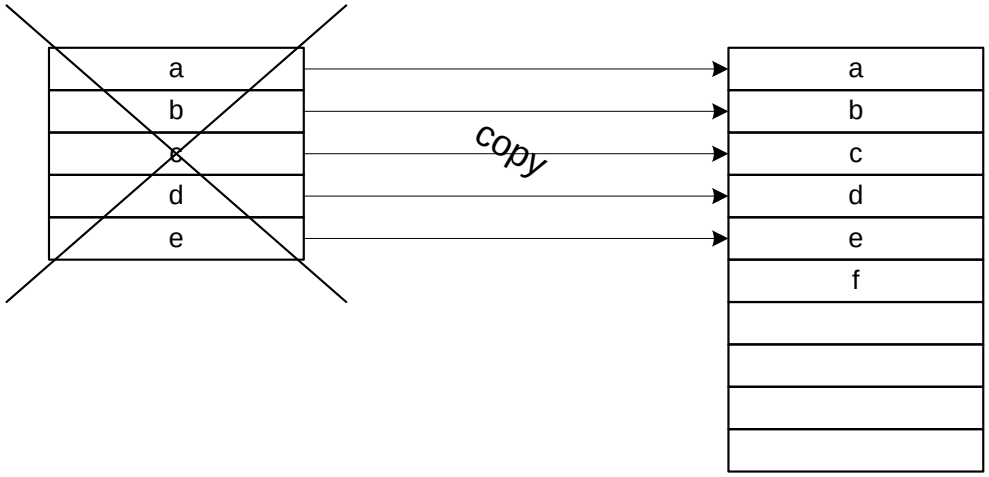
Arrays are a very ***simple collections*** that are provided by most languages

- Limited by its fixed size
- Fixed size also allows the array to be ***stored contiguously*** (in one piece)
  - ***Improved efficiency for direct access***

Collections

Overcoming the fixed size of an array is not particularly difficult

1. *Create a new array*
2. *Copy the elements across to the new array*
3. *Discard the old array*



- ***This is how the List<> generic collection works!***

Collections

The Microsoft.Net framework provides a number of ***generic collections***, for which we examine the List<> type

using System.Collections.Generic;

Declaration:

```
List<type> name = new List<type>();
```

- Example:

```
List<int> numberList = new List<int>();
List<Person> Sit232List = new List<Person>();
```

Collections

Useful **features** of the List<> type:

- **Properties**:
  - **Count** – the total number of elements in the list;
- **Methods**:
  - **Add(element)** – store an element at the end of the collection;
  - **Clear()** – remove all elements in the collection/reset the collection;
  - **Contains(element)** – searches the collection for the element and returns true if it's found (false otherwise);
  - **Insert(index, element)** – inserts an element into the list at the indicated index;
  - **Remove(element)** – searches the collection for the element and removes it from the list if found, returning true if the element was found and removed (false otherwise);
  - **RemoveAt(index)** – removes the element from the collection that is stored at the specified index;
  - **Sort()** – sorts the elements stored in the list in ascending order;

Collections

We have *previous examined repetition control structures*:

- **while**
- **do-while**
- **for**

There is also the ***foreach loop***, a loop specifically for working with collections:

```
foreach(type name in collection_name)
        statement;
```

Method overloading

Method overloading refers to having *several methods with the same name*

Such *methods are differentiated by their signature*

- **Number of parameters**
- **Data type of parameters**
- **How they are passed (input/reference/output)**

Method overloading

Method overloading can ***improve code readability***, e.g.,

```
Network.SendInt32(123);
Network.SendFloat(1.23);
Network.SendString("123");
```

- Becomes

```
Network.Send(123);
Network.Send(1.23);
Network.Send("123");
```

However, this comes at the expense that for every new data type to be supported a *new implementation/method overload must be added*.

- ***Generics provide a good solution to this problem, expected in Session 10.***

Recursion

Recursion is an alternative mechanism through which repetition can be achieved

- Iteration typically applies some *algorithm to different data repetitively* (while/do-while/for/foreach)

- Recursion is where an algorithm includes an invocation to itself
  - i.e., the solution to a problem is expressed in terms of a solution to the same problem, only smaller

Recursion

The design of a recursive algorithm is critical

- It is very easy to develop a recursive algorithm that never terminates

There are two parts to any recursive algorithm

- The base case that solves the problem
  - e.g., Factorial(0) = 1
- The general case that reduces the size of the problem
  - e.g., Factorial(n) = n * Factorial(n – 1)

Each invocation of *the function should progress (general case) towards the termination condition (base case)*

Recursion

Example: Factorial

- Factorial is a mathematical operation in which *all numbers, up to the specified input number, are multiplied*, e.g.,
  - 10! = 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1
- The factorial problem is recursive, as shown on the following slides

Recursion

0! = 1

1! = 1

2! = 2 * 1

3! = 3 * 2 * 1

4! = 4 * 3 * 2 * 1

5! = 5 * 4 * 3 * 2 * 1

6! = 6 * 5 * 4 * 3 * 2 * 1

and so on…

Recursion

$$0! = 1$$

$$1! = 1 = 1 * 0!$$

$$2! = 2 * 1 = 2 * 1!$$

$$3! = 3 * 2 * 1 = 3 * 2!$$

$$4! = 4 * 3 * 2 * 1 = 4 * 3!$$

$$5! = 5 * 4 * 3 * 2 * 1 = 5 * 4!$$

$$6! = 6 * 5 * 4 * 3 * 2 * 1 = 6 * 5!$$

and so on…

Recursion

Iterative solution to factorial:

```
ulong Factorial(ulong Number)
{
    ulong Result;

    if(Number == 0)  // terminate condition or base case
        Result = 1;
    else
    {   // general case or processing loop
        Result = Number; // pre-condition of the loop
        while(--Number != 0) // no negatives possible
            Result *= Number;
    }

    return Result;
}
```

Recursion

Recursive solution to factorial:

```
ulong Factorial(ulong Number) // Number = 10
{
    ulong Result;

    // base case
    if(Number == 0) // no negatives possible
        Result = 1;

    else // general case to 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1
        Result = Number * Factorial(Number - 1); // general case

    return Result; // 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 * 1 (for 0 value)
}
```

Recursion

Advantages of recursion

- ***Some algorithms can be stated simpler using***, or are more naturally stated using, a recursive algorithm
- ***Some data structures are recursive***, thus recursive algorithms are more natural for these structures
- ***Recursion provides the backtracking ability***, e.g., for path finding in games

Disadvantages of recursion

- Recursive algorithms are generally less efficient than the equivalent iterative one (holds both ways)
- Recursive algorithms require more memory
  - Each function call takes memory for parameters, local variables, return values, and temporary data

Recursion

There are *many problems in real life that are suitable for recursion*

- **Mathematical**
  - Factorial
  - Fibonacci
  - Global Common Denominator (GCD)
  - Fourier transform
- **Games**
  - Towers of Hanoi
  - Tic Tac Toe
  - Chess
  - Path finding

Argument promotion and casting

Consider the following two statements:

```
int wholeNumber = 5.0 * 2;
double realNumber = 5.0 * 2;
```

The first produces the compile error:

> Cannot implicitly convert type 'double' to 'int'. An *explicit conversion* exists (are you *missing a cast*?)

Why?

Argument promotion and casting

The answer is found starting with literals, i.e.,

```
int wholeNumber = 5.0 * 2;
double realNumber = 5.0 * 2;
```

Becomes:

```
int wholeNumber = double * integer;
double realNumber = double * integer;
```

The CPU cannot multiply different data types, thus the integer is "promoted" to a double, i.e.,

```
int wholeNumber = double * double;
double realNumber = double * double;
```

The result of double * double is also a double, i.e.,

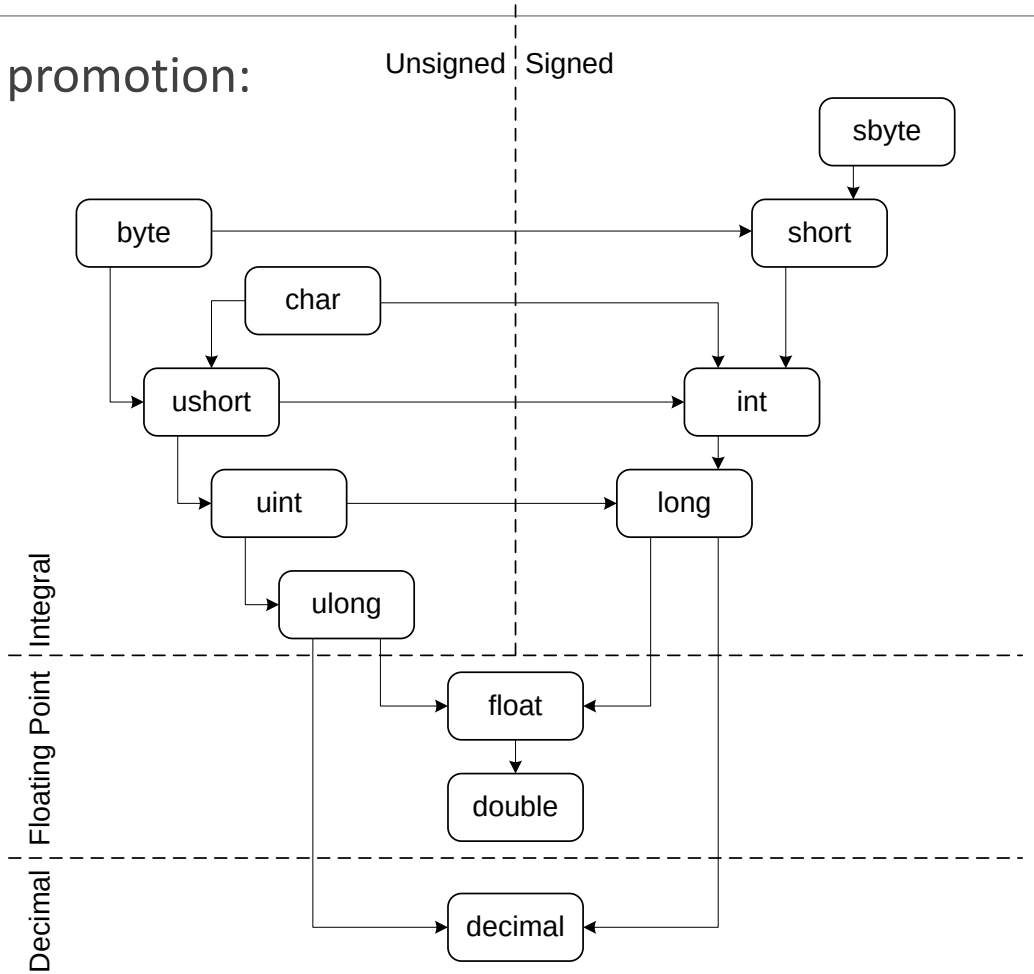
```
int wholeNumber = double;
double realNumber = double;
```

Now the error makes sense!

- Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)

Argument promotion and casting

Rules for argument promotion:



Argument promotion and casting

The problem is **solved by converting** the result back to an integer, i.e.,

```
int wholeNumber = Convert.ToInt32(5.0 * 2);
```

Alternatively, we can also use casting

```
int wholeNumber = (int)(5.0 * 2);
```

Differences:

- Convert supports string to numeric conversions
- Casting will not produce any feedback if data is lost, whereas Convert will throw an exception
- Casting is also used for Inheritance and Polymorphism

Validity checking

Validity checking refers to *checking that a user's input is valid*

- Becoming a *fundamental requirement for all developers* due to security attacks

General approach:

- Assume all input is **unsafe**
- **Only** accept ***inputs known to be safe***

Validity checking

We consider only two simple checks at this time:

- Make sure ***data input in the correct type***, i.e., only digits (0-9), '-' (negative), or '.' (decimal place) are entered for a numeric value;

- Values ***should be in a correct range***, both for the problem, e.g., the 30th and 31st of February do not exist, and appropriate for any data storage, e.g., *storing 200 in an sbyte variable actually stores -56*;

Validity checking

Some useful static methods defined by char:

- **char.IsDigit** – tests for a decimal digit (0..9);
- **char.IsLetter** – tests for a letter (a..zA..z);
- **char.IsLetterOrDigit** – tests for either a letter or digit (a..zA..Z0..9);
- **char.IsLower** – tests for a lower case letter (a..z);
- **char.IsNumber** – tests for a number (0..9 and also numbers in other character sets);
- **char.IsPunctuation** – tests for punctuation (. , ! ? etc.);
- **char.IsSeparator** – tests for a separator (space and new line);
- **char.IsSymbol** – tests for symbols ($ + – etc.)
- **char.IsUpper** – tests for an upper case letter (A..Z); and
- **char.IsWhiteSpace** – tests for a white space character (space, tab, and new line).

Summary

Session 03. Methods and arrays

- Objectives
- Collections
- Method overloading
- Recursion
- Argument promotion and casting
- Validity checking

Summary

Training Videos:

- C#: Arrays
- VS: Command Line Arguments
- C#: Static Members
- C#: Parameter Passing