Sarah Coufal
Elly Fernández
Kejing Wang
Ziyao Wang

C# has built-in types to represent numbers (integers, floating point values), as well as Boolean expressions and text characters.

There are several different integral types in C#: **sbyte** (signed 8-bit), **byte** (unsigned 8-bit), **short** (signed 16-bit), **ushort** (unsigned 16-bit), **int** (signed 32-bit), **uint** (unsigned 32-bit), **long** (signed 64-bit), and **ulong** (unsigned 64-bit).The standard operators defined on integers are arithmetic operators (**+,-,*,/,%,++,--,+,-**), bitwise and shift operators (left and right shift, logical operators), comparison operators (**<, >, <=, >=**), and equality operators (**==,!=**).

The built in types **float** (4 bytes), **double** (8 bytes), and **decimal** (16 bytes) are used to define floating point values. These types support arithmetic, comparison, and equality operators.

Boolean values (**true, false**) are represented by the **bool** type (1 byte). This type is the result of comparison and equality operators. The standard operators for the **bool** type are the logical operators (negation **!**, logical AND **&**, logical OR **|**, logical exclusive OR **^**, conditional logical AND **&&**, and condition logical OR **||**). Unlike in the C and C++ languages, there are no conversions between the **bool** type and other types, and the **bool** type is distinct from integral values.

The **char** type (16-bit) represents a Unicode UTF-16 character. This type supports comparison, equality, increment, and decrement operators. Additionally, the arithmetic and bitwise logical operators can be performed on char operands using their corresponding character codes and producing an int type result.

(Variable scope, type inference, LINQ)

- All variables are lexically scoped in C#.
- The default type declaration in C# is explicit. Without type inference, we have to explicitly define what would be a string, an integer, or any other type before compile time. With type inference, if we do not want to decide beforehand what the type should be, we can declare a variable using "var." This variable is still strongly typed, it is just the compiler that determines the type not the programmer.
- The above feature can be easily integrated with Language-Integrated Query (LINQ). LINQ is a set of query capabilities very similar to SQL queries, and the goal is to filter out data based on specific criteria. LINQ supports most data types, including XML, SQL, etc.

(Variable Name and Declaration)

C# is flexible with naming. As C# official language documentation specifies for identifiers, all identifiers must start with a letter or "_", and may contain Unicode letters, numbers, and even Unicode connecting or formatting characters. Conflicts with reserved keywords can also be resolved by using "@" in front of the identifier. Other than these rules, no constraints are clearly specified on naming. However, to be consistent with the Microsoft's .NET

Framework and general convention, several rules for local variables and method arguments are suggested as below:

1. Use camelCasing for local variables and method arguments
2. Use only [A-z] and [0, 9], even though C# supports unicode charset
3. Using '_' as the first letter of variable name is not advised.
4. Including '_', in the variable name is not advised.

There are multiple ways to declare a variable in C#: you can either explicitly specify the type as you do in Java, or use the var keyword to let the compiler infer the type:

1. Explicitly-typed variable: can be initialized when it's declared or later,
   <type> <name>;
   <type> <name> = <value>;
   E.g. int i;
   E.g. int i = 10;

2. Implicitly typed variable using "var" keyword: must be initialized when it's declared.
   var <name> = <value>;
   E.g. var a = "Hello";

3. Multiple declarations on a single line only works for explicit types:
   <type> <name1>, <name2>, ...;
   E.g. int a, b;
   Initialize any variables of your choice:
   E.g. int a = 1, b, c = 3, d, e, f;

(Control Structures: Assignment and Multiple Assignment)

The assignment in C# is similar to most modern programming languages, using '=' as assignment operator:

Assignment: **<var name> = <expression>;**
E.g. person = "Joe";

C# supports combination assignment operators:
x op= y
C#'s combination assignment is supported by:
Arithmetic operators: **+=, -=, /=, *=, %=**
Boolean logical operators: **&=, |=, ^=**
Bitwise operators and shift: **&=, |=, ^=, <<=, >>=**
C# also support pre/post increment/decrement operators: **++/--**

However, C# doesn't support multiway or parallel assignment like those Python does, e.g. x, y = y, x. For example, in C# a tuple gives rise to multiple return values from a method call, but once you retrieve the tuple, you have to handle the value through individual assignments.

```
using System;
```

```
public class Example
{
    public static void Main()
    {
        var result = QueryCityData("New York City");

        var city = result.Item1;
        var pop = result.Item2;
        var size = result.Item3;

         // Do something with the data.
    }

    private static (string, int, double) QueryCityData(string name)
    {
        if (name == "New York City")
            return (name, 8175133, 468.48);

        return ("", 0, 0);
    }
}
```

Starting with C# 7, there is the tuple-deconstruction feature which enables the multiple assignments of values of a tuple to different variables in one single line, it's still an improvement about tuples and not the assignment feature of language itself.

Conditional Statements--if/else

```
if (<condition>)
      <statement>;
else if (<condition>)
      <statement>;
else
      <statement>;
```

The conditions for `if` clauses must evaluate to a `bool` value
C# is not white space sensitive, so the indentation in conditionals is not crucial for execution. However, this means that curly braces must be used when you want to have more than one statement in the `if` or `else` clause.

```
if (<condition>)
{
      <statement(s)>;
}
```

```
else if (<condition>)
{
      <statement(s)>;
}
else
{
      <statement(s)>;
}

bool haveFever = true;
if (haveFever)
{
      Console.WriteLine("Stay home!");
}
else
{
      Console.WriteLine("Go to class!");
}
```

Conditional Statements--switch/case

- The switch statement is a selection statement that only executes one switch section from the list based on a pattern match to the match expression -> `switch (expr)`
    - The match expression must return a `char, string, bool, integral value (int or long), or enum`
- There can be any number of switch sections, and each section can have one or more case labels, but no two case labels can have the same expression.
- A compiler error will be generated if you try to "fall through" the switch sections; thus, the break statement must be used.

```
switch (caseSwitch)
{
      case 1:
            <statement>;
            break;
      case 2:
            <statement>;
            break;
      default:
            <statement>;
            break;
}

Random rnd = new Random();
```

```
int caseSwitch = rnd.Next(1, 2);
switch (caseSwitch)
{
      case 1:
            Console.WriteLine(“Heads”);
            break;
      default:
            Console.WriteLine(“Tails”)
            break;
}
```

Loops--while

- Executes zero or more times

```
while (<condition>)
{
      <statement(s)>;
}
```

```
int counter = 0;
while (counter < 5)
{
      Console.WriteLine($”{counter}\n”);
      counter++;
}
Console.WriteLine(“BLAST OFF!!!”);
```

- Note: this print statement uses string interpolation, identified by the `$` special character. The value(s) within curly braces in string interpolation will be repealed by the string representations of the expression results. This feature starters with C# 6.

Loops--do/while

- Executes one or more times
- `do` executes one or more statements that evaluate to `true`

```
do
{
      <statement(s)>;
} while (<condition>);
```

```
int counter = 0;
do
{
      Console.WriteLine($“{counter}”);
```

```
        counter++;
    } while (counter < 5);
    Console.WriteLine(“BLAST OFF!!!”);
```

Loops--for

```
    for(<initializer>; <condition>; <iterator>)
    {
        <statement(s)>;
    }
```

- The initializer must be the declaration and initialization of a local loop variable (with a scope limited to within the loop), an assignment, invocation of a method, increments or decrements, creation of an object, or an await expression.
- The condition must evaluate to a boolean; if the value is false, the loop is exited.
- Like the initializer, the iterator can be an assignment statement, invocation of a method, increment, decrement, create of an object, or an await expression[1]

```
    for (int i = 5; i >= 0; i--)
    {
        Console.WriteLine($“{i}”);
    }
    Console.WriteLine(“BLAST OFF!!!”);
```

Loops--for each

- Can be used with an instance of any type that has a public parameterless `GetEnumerator` method with a return type of a class, struct, or interface. This return type of the `GetEnumerator` method must have the public `Current` property and the public, parameterless `MoveNext` method (that returns a `bool`)
- Executes one or more statements for every element in a collection

```
    // using var means the compiler interprets the type
    foreach (var <item> in <collection>)
    {
        <statement(s)>;
    }

    var timesTwo = new List<int> {0, 1, 2, 3, 4};
    foreach (int element in timesTwo)
    {
        element = element * 2;
    }
```

[1] Await operator is used with asynchronous programming

Jump Statements

- `break` statements terminate the closest enclosing loop (control is returned only one level up in nested loops) or switch statement.
- `continue` statements pass control to the next iteration of the enclosing `while, do, for,` or `foreach` statement in which it appears.
- `goto` statements direct the program control directly to a labeled statement; often these statements are used to jump between cases in a switch

Works Cited

Built-in Types (C# Reference)

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/built-in-types

Identifiers and Naming Conventions(C# lang specification, .Net Framework Documentation)

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/identifier-names

https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/general-naming-conventions?redirected from=MSDN

Local Variable Declaration (C# lang specification, tutorial):

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/implicitly-typed-local-variables

https://www.dotnetperls.com/multiple-local-variable

Control Structure:

Assignment

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/assignment-operator

https://docs.microsoft.com/en-us/dotnet/csharp/deconstruct

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators#increment-operator-

String Interpolation

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated

Conditionals and Loops

https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/intro-to-csharp/branches-and-loops-local

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/foreach-in#code-try-0

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/switch

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/continue

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/goto