C# Foundation Level - Complete Guide with Examples

1. Variables and Data Types

Concept Explanation

Variables are containers that store data values. In C#, every variable has a specific data type that determines what kind of data it can hold and how much memory it occupies.

Common Data Types:

- **int**: 32-bit signed integer (-2,147,483,648 to 2,147,483,647)
- **string**: Sequence of characters (text)
- **bool**: Boolean value (true or false)
- **double**: 64-bit floating-point number
- **float**: 32-bit floating-point number
- **decimal**: 128-bit decimal number (high precision for financial calculations)
- **char**: Single Unicode character
- **byte**: 8-bit unsigned integer (0 to 255)
- **long**: 64-bit signed integer

Sample Program:

```csharp
```

```
using System;

class VariablesAndDataTypes
{
    static void Main()
    {
        // Integer types
        int age = 25;
        long population = 7800000000L;
        byte score = 95;

        // Floating-point types
        double price = 19.99;
        float temperature = 36.5f;
        decimal salary = 75000.50m;

        // Text types
        string name = "John Doe";
        char grade = 'A';

        // Boolean type
        bool isStudent = true;
        bool isEmployed = false;

        // Display all variables
        Console.WriteLine("=== Variables and Data Types ===");
        Console.WriteLine($"Name: {name}");
        Console.WriteLine($"Age: {age}");
        Console.WriteLine($"Population: {population}");
        Console.WriteLine($"Score: {score}");
        Console.WriteLine($"Price: {price}");
        Console.WriteLine($"Temperature: {temperature}");
        Console.WriteLine($"Salary: {salary}");
        Console.WriteLine($"Grade: {grade}");
        Console.WriteLine($"Is Student: {isStudent}");
        Console.WriteLine($"Is Employed: {isEmployed}");

        // Getting type information
        Console.WriteLine($"\nType of age: {age.GetType()}");
        Console.WriteLine($"Type of name: {name.GetType()}");
        Console.WriteLine($"Type of price: {price.GetType()}");
```

```
    }
}
```

---

2. Basic Operators

Concept Explanation

Operators are symbols that perform operations on variables and values.

Types of Operators:

Arithmetic Operators

- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `%` Modulus (remainder)
- `++` Increment
- `--` Decrement

Comparison Operators

- `==` Equal to
- `!=` Not equal to
- `>` Greater than
- `<` Less than
- `>=` Greater than or equal to
- `<=` Less than or equal to

Logical Operators

- `&&` Logical AND
- `||` Logical OR
- `!` Logical NOT

Sample Program:

```csharp
```

```
using System;

class BasicOperators
{
    static void Main()
    {
        Console.WriteLine("=== Arithmetic Operators ===");
        int a = 15;
        int b = 4;

        Console.WriteLine($"a = {a}, b = {b}");
        Console.WriteLine($"Addition: {a} + {b} = {a + b}");
        Console.WriteLine($"Subtraction: {a} - {b} = {a - b}");
        Console.WriteLine($"Multiplication: {a} * {b} = {a * b}");
        Console.WriteLine($"Division: {a} / {b} = {a / b}");
        Console.WriteLine($"Modulus: {a} % {b} = {a % b}");

        // Increment and Decrement
        int x = 10;
        Console.WriteLine($"\nOriginal x: {x}");
        Console.WriteLine($"Pre-increment (++x): {++x}");
        Console.WriteLine($"Post-increment (x++): {x++}");
        Console.WriteLine($"After post-increment: {x}");
        Console.WriteLine($"Pre-decrement (--x): {--x}");
        Console.WriteLine($"Post-decrement (x--): {x--}");
        Console.WriteLine($"After post-decrement: {x}");

        Console.WriteLine("\n=== Comparison Operators ===");
        int num1 = 20;
        int num2 = 15;

        Console.WriteLine($"num1 = {num1}, num2 = {num2}");
        Console.WriteLine($"num1 == num2: {num1 == num2}");
        Console.WriteLine($"num1 != num2: {num1 != num2}");
        Console.WriteLine($"num1 > num2: {num1 > num2}");
        Console.WriteLine($"num1 < num2: {num1 < num2}");
        Console.WriteLine($"num1 >= num2: {num1 >= num2}");
        Console.WriteLine($"num1 <= num2: {num1 <= num2}");

        Console.WriteLine("\n=== Logical Operators ===");
        bool condition1 = true;
        bool condition2 = false;
```

```
        Console.WriteLine($"condition1 = {condition1}, condition2 = {condition2}");
        Console.WriteLine($"condition1 && condition2: {condition1 && condition2}");
        Console.WriteLine($"condition1 || condition2: {condition1 || condition2}");
        Console.WriteLine($"!condition1: {!condition1}");
        Console.WriteLine($"!condition2: {!condition2}");

        // Practical example
        int age = 25;
        bool hasLicense = true;
        bool canDrive = (age >= 18) && hasLicense;
        Console.WriteLine($"\nAge: {age}, Has License: {hasLicense}");
        Console.WriteLine($"Can Drive: {canDrive}");
    }
}
```

---

3. Console Input/Output

Concept Explanation

Console I/O allows your program to interact with users through the command line interface.

- **Console.WriteLine()**: Outputs text to console and moves to next line
- **Console.Write()**: Outputs text to console without moving to next line
- **Console.ReadLine()**: Reads a line of text input from user
- **Console.ReadKey()**: Reads a single key press

Sample Program:

csharp

```
using System;

class ConsoleInputOutput
{
    static void Main()
    {
        Console.WriteLine("=== Console Input/Output Demo ===");

        // Basic output
        Console.WriteLine("Hello, World!");
        Console.Write("This is on the same line. ");
        Console.WriteLine("This continues the line.");

        // Getting user input
        Console.Write("Enter your name: ");
        string userName = Console.ReadLine();

        Console.Write("Enter your age: ");
        string ageInput = Console.ReadLine();
        int userAge = int.Parse(ageInput);

        Console.Write("Are you a student? (true/false): ");
        string studentInput = Console.ReadLine();
        bool isStudent = bool.Parse(studentInput);

        // Display collected information
        Console.WriteLine("\n=== Your Information ===");
        Console.WriteLine($"Name: {userName}");
        Console.WriteLine($"Age: {userAge}");
        Console.WriteLine($"Student: {isStudent}");

        // Formatted output examples
        Console.WriteLine("\n=== Formatted Output Examples ===");
        double price = 99.99;
        Console.WriteLine("Price: ${0}", price);
        Console.WriteLine($"Price with string interpolation: ${price}");
        Console.WriteLine("Price with formatting: {0:C}", price); // Currency format
        Console.WriteLine("Price with 2 decimals: {0:F2}", price);

        // Wait for key press
        Console.WriteLine("\nPress any key to exit...");
        Console.ReadKey();
```

```
    }
}
```

---

4. Comments and Code Documentation

Concept Explanation

Comments are non-executable text in your code that explain what the code does. They help other developers (and future you) understand the code.

Types of Comments:

- **Single-line comments**: `// Comment text`
- **Multi-line comments**: `/* Comment text */`
- **XML documentation comments**: `/// <summary>Description</summary>`

Sample Program:

```csharp
```

```csharp
using System;

/// <summary>
/// This class demonstrates different types of comments and documentation
/// in C# programming.
/// </summary>
class CommentsAndDocumentation
{
    /// <summary>
    /// Main method - entry point of the program
    /// </summary>
    /// <param name="args">Command line arguments</param>
    static void Main(string[] args)
    {
        // This is a single-line comment
        // It explains what the next line does
        Console.WriteLine("=== Comments and Documentation Demo ===");

        /*
         * This is a multi-line comment
         * It can span multiple lines
         * Useful for longer explanations
         */

        // Variable declarations with explanatory comments
        int studentCount = 25;    // Number of students in class
        double averageGrade = 87.5; // Class average grade

        /*
         * Calculate and display student statistics
         * This section performs basic calculations
         */
        double totalPoints = studentCount * averageGrade;

        // Display results with descriptive comments
        Console.WriteLine($"Students: {studentCount}"); // Show student count
        Console.WriteLine($"Average: {averageGrade}");  // Show average grade
        Console.WriteLine($"Total Points: {totalPoints}"); // Show total points

        // TODO: Add more statistical calculations
        // FIXME: Handle division by zero in future calculations
        // NOTE: Consider adding input validation
```

```
        DisplayWelcomeMessage(); // Call helper method
    }

    /// <summary>
    /// Displays a welcome message to the user
    /// </summary>
    /// <remarks>
    /// This method demonstrates XML documentation comments
    /// which can be used to generate API documentation
    /// </remarks>
    static void DisplayWelcomeMessage()
    {
        Console.WriteLine("\nWelcome to C# Programming!");
        Console.WriteLine("Comments make code more readable and maintainable.");
    }
}
```

---

5. Basic String Operations

Concept Explanation

Strings are sequences of characters used to represent text. C# provides many built-in methods to manipulate strings.

Common String Operations:

- **Length**: Get string length
- **Concatenation**: Combine strings
- **Substring**: Extract part of string
- **ToUpper/ToLower**: Change case
- **Trim**: Remove whitespace
- **Replace**: Replace characters/substrings
- **Split**: Split string into array
- **Contains**: Check if string contains substring

Sample Program:

csharp

```
using System;

class BasicStringOperations
{
    static void Main()
    {
        Console.WriteLine("=== Basic String Operations ===");

        // String declaration and initialization
        string firstName = "John";
        string lastName = "Doe";
        string email = "  john.doe@email.com  ";
        string sentence = "The quick brown fox jumps over the lazy dog";

        // String concatenation
        string fullName = firstName + " " + lastName;
        string greeting = $"Hello, {fullName}!"; // String interpolation

        Console.WriteLine($"First Name: {firstName}");
        Console.WriteLine($"Last Name: {lastName}");
        Console.WriteLine($"Full Name: {fullName}");
        Console.WriteLine($"Greeting: {greeting}");

        // String properties and methods
        Console.WriteLine("\n=== String Properties and Methods ===");
        Console.WriteLine($"Length of full name: {fullName.Length}");
        Console.WriteLine($"Uppercase: {fullName.ToUpper()}");
        Console.WriteLine($"Lowercase: {fullName.ToLower()}");

        // Working with email string
        Console.WriteLine($"\nOriginal email: '{email}'");
        string cleanEmail = email.Trim(); // Remove leading/trailing spaces
        Console.WriteLine($"Trimmed email: '{cleanEmail}'");

        // Substring operations
        string domain = cleanEmail.Substring(cleanEmail.IndexOf('@') + 1);
        string username = cleanEmail.Substring(0, cleanEmail.IndexOf('@'));
        Console.WriteLine($"Username: {username}");
        Console.WriteLine($"Domain: {domain}");

        // String searching and checking
        Console.WriteLine($"\n=== String Searching ===");
        Console.WriteLine($"Email contains 'john': {cleanEmail.Contains("john")}");
```

```
        Console.WriteLine($"Email starts with 'john': {cleanEmail.StartsWith("john")}");
        Console.WriteLine($"Email ends with '.com': {cleanEmail.EndsWith(".com")}");

        // String replacement
        string modifiedSentence = sentence.Replace("fox", "cat");
        Console.WriteLine($"\nOriginal: {sentence}");
        Console.WriteLine($"Modified: {modifiedSentence}");

        // String splitting
        string[] words = sentence.Split(' ');
        Console.WriteLine($"\nSentence has {words.Length} words:");
        foreach (string word in words)
        {
            Console.WriteLine($"- {word}");
        }

        // String comparison
        string str1 = "Hello";
        string str2 = "hello";
        Console.WriteLine($"\n=== String Comparison ===");
        Console.WriteLine($"'{str1}' == '{str2}': {str1 == str2}");
        Console.WriteLine($"Case-insensitive comparison: {str1.Equals(str2, StringComparison.OrdinalIgnoreCase)}");
    }
}
```

---

6. Type Conversion and Casting

Concept Explanation

Type conversion is the process of converting one data type to another. There are two types:

- **Implicit Conversion**: Automatic conversion (safe, no data loss)
- **Explicit Conversion**: Manual conversion (casting, potential data loss)

Conversion Methods:

- **Parse()**: Convert string to specific type
- **Convert class**: Universal conversion methods
- **TryParse()**: Safe parsing with error handling
- **Cast operator**: (type) for explicit casting

Sample Program:

csharp

```
using System;

class TypeConversionAndCasting
{
    static void Main()
    {
        Console.WriteLine("=== Type Conversion and Casting ===");

        // Implicit conversion (widening)
        Console.WriteLine("=== Implicit Conversion ===");
        int intValue = 100;
        long longValue = intValue;     // int to long (safe)
        float floatValue = intValue;   // int to float (safe)
        double doubleValue = floatValue; // float to double (safe)

        Console.WriteLine($"int: {intValue}");
        Console.WriteLine($"long: {longValue}");
        Console.WriteLine($"float: {floatValue}");
        Console.WriteLine($"double: {doubleValue}");

        // Explicit conversion (casting)
        Console.WriteLine("\n=== Explicit Conversion (Casting) ===");
        double largeNumber = 123.456;
        int truncatedInt = (int)largeNumber;  // Data loss - decimal part removed
        float smallerFloat = (float)largeNumber;

        Console.WriteLine($"Original double: {largeNumber}");
        Console.WriteLine($"Casted to int: {truncatedInt}");
        Console.WriteLine($"Casted to float: {smallerFloat}");

        // String to number conversion using Parse
        Console.WriteLine("\n=== String to Number Conversion (Parse) ===");
        string numberString = "456";
        string decimalString = "78.9";
        string boolString = "true";

        int parsedInt = int.Parse(numberString);
        double parsedDouble = double.Parse(decimalString);
        bool parsedBool = bool.Parse(boolString);

        Console.WriteLine($"String '{numberString}' to int: {parsedInt}");
        Console.WriteLine($"String '{decimalString}' to double: {parsedDouble}");
        Console.WriteLine($"String '{boolString}' to bool: {parsedBool}");
```

```
// Safe parsing with TryParse
Console.WriteLine("\n=== Safe Parsing with TryParse ===");
string validNumber = "123";
string invalidNumber = "abc";

// TryParse returns bool indicating success/failure
if (int.TryParse(validNumber, out int result1))
{
    Console.WriteLine($"Successfully parsed '{validNumber}' to: {result1}");
}

if (int.TryParse(invalidNumber, out int result2))
{
    Console.WriteLine($"Successfully parsed '{invalidNumber}' to: {result2}");
}
else
{
    Console.WriteLine($"Failed to parse '{invalidNumber}' - not a valid integer");
}

// Using Convert class
Console.WriteLine("\n=== Using Convert Class ===");
string convertString = "789";
double convertDouble = 45.67;
bool convertBool = true;

int convertedInt = Convert.ToInt32(convertString);
string convertedString = Convert.ToString(convertDouble);
int convertedFromBool = Convert.ToInt32(convertBool); // true = 1, false = 0

Console.WriteLine($"Convert string to int: {convertedInt}");
Console.WriteLine($"Convert double to string: '{convertedString}'");
Console.WriteLine($"Convert bool to int: {convertedFromBool}");

// Demonstration of overflow in casting
Console.WriteLine("\n=== Overflow in Casting ===");
int largeInt = 300;
byte smallByte = (byte)largeInt; // byte range: 0-255
Console.WriteLine($"Large int {largeInt} casted to byte: {smallByte}");
Console.WriteLine("Notice the overflow - value wrapped around!");

// Character to number conversion
Console.WriteLine("\n=== Character Conversions ===");
```

```
        char digitChar = '5';
        char letterChar = 'A';

        int digitValue = (int)digitChar; // ASCII value
        int letterValue = (int)letterChar; // ASCII value
        int numericValue = digitChar - '0'; // Convert char digit to actual number

        Console.WriteLine($"Character '{digitChar}' ASCII value: {digitValue}");
        Console.WriteLine($"Character '{letterChar}' ASCII value: {letterValue}");
        Console.WriteLine($"Character '{digitChar}' numeric value: {numericValue}");
    }
}
```

---

7. Constants and Literals

Concept Explanation

Constants are fixed values that cannot be changed during program execution. Literals are the actual values assigned to variables or constants.

Types of Constants:

- **const**: Compile-time constant
- **readonly**: Runtime constant
- **static readonly**: Class-level runtime constant

Types of Literals:

- **Integer literals**: 42, 100L, 0xFF
- **Floating-point literals**: 3.14, 2.5f, 1.5m
- **Boolean literals**: true, false
- **Character literals**: 'A', '\n'
- **String literals**: "Hello", @"C:\Path"
- **Null literal**: null

Sample Program:

csharp

```
using System;

class ConstantsAndLiterals
{
    // Class-level constants
    const double PI = 3.14159265359;
    const int MAX_STUDENTS = 30;
    const string SCHOOL_NAME = "Tech Academy";

    // readonly can be assigned at runtime in constructor
    static readonly DateTime PROGRAM_START_TIME = DateTime.Now;
    readonly int instanceId;

    // Constructor to demonstrate readonly
    public ConstantsAndLiterals()
    {
        instanceId = new Random().Next(1000, 9999);
    }

    static void Main()
    {
        Console.WriteLine("=== Constants and Literals Demo ===");

        // Using class constants
        Console.WriteLine($"School: {SCHOOL_NAME}");
        Console.WriteLine($"Maximum students: {MAX_STUDENTS}");
        Console.WriteLine($"PI value: {PI}");
        Console.WriteLine($"Program started at: {PROGRAM_START_TIME}");

        // Local constants
        const int DAYS_IN_WEEK = 7;
        const string GREETING = "Welcome";

        Console.WriteLine($"\nLocal constants:");
        Console.WriteLine($"Days in week: {DAYS_IN_WEEK}");
        Console.WriteLine($"Greeting: {GREETING}");

        // Integer literals
        Console.WriteLine("\n=== Integer Literals ===");
        int decimalNumber = 100;        // Decimal literal
        int hexNumber = 0xFF;           // Hexadecimal literal (255)
        int binaryNumber = 0b1010;      // Binary literal (10)
        long longNumber = 1000000L;     // Long literal
```

```
Console.WriteLine($"Decimal: {decimalNumber}");
Console.WriteLine($"Hexadecimal 0xFF: {hexNumber}");
Console.WriteLine($"Binary 0b1010: {binaryNumber}");
Console.WriteLine($"Long: {longNumber}");

// Floating-point literals
Console.WriteLine("\n=== Floating-Point Literals ===");
float floatLiteral = 3.14f;      // Float literal
double doubleLiteral = 2.71828;  // Double literal (default)
decimal decimalLiteral = 99.99m; // Decimal literal

Console.WriteLine($"Float: {floatLiteral}");
Console.WriteLine($"Double: {doubleLiteral}");
Console.WriteLine($"Decimal: {decimalLiteral}");

// Boolean literals
Console.WriteLine("\n=== Boolean Literals ===");
bool isTrue = true;
bool isFalse = false;

Console.WriteLine($"True literal: {isTrue}");
Console.WriteLine($"False literal: {isFalse}");

// Character literals
Console.WriteLine("\n=== Character Literals ===");
char letter = 'A';
char digit = '5';
char newline = '\n';
char tab = '\t';
char backslash = '\\';
char singleQuote = '\'';

Console.WriteLine($"Letter: {letter}");
Console.WriteLine($"Digit: {digit}");
Console.Write("Before newline");
Console.Write(newline);
Console.WriteLine("After newline");
Console.WriteLine($"Tab:{tab}Example with tab");
Console.WriteLine($"Backslash: {backslash}");
Console.WriteLine($"Single quote: {singleQuote}");

// String literals
Console.WriteLine("\n=== String Literals ===");
```

```
        string regularString = "Hello, World!";
        string stringWithEscape = "Line 1\nLine 2\tTabbed";
        string verbatimString = @"C:\Users\Username\Documents";
        string interpolatedString = $"Current time: {DateTime.Now}";

        Console.WriteLine($"Regular string: {regularString}");
        Console.WriteLine($"With escape characters:\n{stringWithEscape}");
        Console.WriteLine($"Verbatim string: {verbatimString}");
        Console.WriteLine($"Interpolated string: {interpolatedString}");

        // Null literal
        Console.WriteLine("\n=== Null Literal ===");
        string nullString = null;
        int? nullableInt = null;  // Nullable integer

        Console.WriteLine($"Null string: {nullString ?? "null"}");
        Console.WriteLine($"Nullable int: {nullableInt?.ToString() ?? "null"}");

        // Using constants in calculations
        Console.WriteLine("\n=== Using Constants in Calculations ===");
        double radius = 5.0;
        double area = PI * radius * radius;
        double circumference = 2 * PI * radius;

        Console.WriteLine($"Circle with radius {radius}:");
        Console.WriteLine($"Area: {area:F2}");
        Console.WriteLine($"Circumference: {circumference:F2}");

        // Instance example
        var instance = new ConstantsAndLiterals();
        Console.WriteLine($"\nInstance ID (readonly): {instance.instanceId}");

        // Demonstrating constant vs readonly differences
        Console.WriteLine($"\nConst PI: {PI} (compile-time)");
        Console.WriteLine($"Readonly start time: {PROGRAM_START_TIME} (runtime)");
    }
}
```

Summary

These foundation concepts form the building blocks of C# programming:

1. **Variables and Data Types**: Understanding how to store and work with different kinds of data

2. **Basic Operators**: Performing calculations, comparisons, and logical operations
3. **Console I/O**: Interacting with users through input and output
4. **Comments**: Documenting code for better maintainability
5. **String Operations**: Manipulating text data effectively
6. **Type Conversion**: Converting between different data types safely
7. **Constants and Literals**: Working with fixed values and different value representations

Practice Exercises

Try these exercises to reinforce your learning:

1. Create a program that calculates the area and perimeter of different shapes
2. Build a simple calculator using basic operators
3. Make a program that validates user input using type conversion
4. Create a text processing program using string operations
5. Design a program that demonstrates different types of literals and constants

Master these concepts before moving to the next level, as they form the foundation for all advanced C# programming!