C#: Definitive Guide from Beginner to Expert-Level Topics

C# is a modern, type-safe, object-oriented programming language developed by Microsoft, running on the .NET platform. Known for its versatility, C# powers applications from web services (ASP.NET Core) to desktop apps (WinForms, WPF), games (Unity, Godot), and cloud solutions (Azure). This guide is the ultimate resource for C#, covering every topic from beginner to expert level with exhaustive explanations, practical examples, edge cases, and cross-connections, matching the depth of a Rust README.

Table of Contents

1. Introduction to C#
2. Getting Started: Hello, World!
3. C# Versions and .NET
4. Basic Concepts
    - Variables and Data Types
    - Functions and Methods
    - Control Flow
5. Object-Oriented Programming
    - Classes and Objects
    - Inheritance and Interfaces
    - Records
6. Generics
7. Delegates and Events
8. LINQ (Language Integrated Query)
9. Error Handling
10. Asynchronous Programming
11. Collections and Data Structures
12. Advanced Topics
    - Pattern Matching
    - Source Generators
    - Native Interop
    - Memory Management
    - Reflection and Attributes

    - Performance Optimization
    - Security Best Practices
13. Testing in C#
    - xUnit
    - Moq
    - Code Coverage
14. .NET Ecosystem
    - ASP.NET Core
    - Entity Framework Core
    - Blazor
15. Building a Sample Project
16. Resources

Introduction to C#

C#, introduced in 2000 by Microsoft, is a general-purpose language combining C/C++'s performance with Java's simplicity. Its key features include:

- **Type Safety**: Strong typing prevents errors at compile time.
- **Object-Oriented**: Supports classes, interfaces, and inheritance.
- **Modern Features**: LINQ, async/await, records, pattern matching.
- **Cross-Platform**: Runs on Windows, Linux, macOS via .NET Core.
- **Extensive Ecosystem**: Integrates with ASP.NET, Entity Framework, Unity, and Azure.
- **Use Cases**: Web APIs, desktop apps, games, cloud services, IoT, and mobile apps (MAUI).

C#'s popularity stems from its productivity, performance, and integration with the .NET platform, making it ideal for enterprise and hobbyist developers.

Getting Started: Hello, World!

Install the .NET SDK from dotnet.microsoft.com. Verify installation:

```
dotnet --version
```

Create a console app:

```
dotnet new console -n MyApp
cd MyApp
```

Edit `Program.cs`:

```
Console.WriteLine("Hello, World!");
```

Run:

```
dotnet run
```

Use Visual Studio, VS Code (with C# extension), or Rider for development. Initialize a project with NuGet for package management:

```
dotnet new sln
dotnet new console -n MyApp
dotnet sln add MyApp
dotnet add MyApp package Newtonsoft.Json
```

C# Versions and .NET

C# evolves with .NET releases:

- **C# 1–5**: (2002–2012) Introduced generics, LINQ, async/await.
- **C# 6–8**: (2015–2019) Null-conditional operators, pattern matching, records.
- **C# 9–12**: (2020–2023) Init-only properties, primary constructors, file-scoped namespaces.
- **C# 13**: (2024, .NET 9) Field keyword, enhanced params collections, lock improvements.

Key features by version:

- **6.0**: Expression-bodied members, null-conditional (`?.`).
- **7.0**: Tuples, discards, local functions.
- **8.0**: Nullable reference types, default interface methods.
- **9.0**: Records, init-only setters, top-level statements.
- **10.0**: Global using, file-scoped namespaces.
- **11.0**: Raw string literals, UTF-8 strings.
- **12.0**: Primary constructors, collection expressions.
- **13.0**: `field` keyword, `params` with Span<T>, partial properties.

Check version:

```
dotnet --info
```

Target specific versions in `csproj`:

```
<PropertyGroup>
  <TargetFramework>net9.0</TargetFramework>
  <LangVersion>13.0</LangVersion>
</PropertyGroup>
```

Basic Concepts

Variables and Data Types

C# is strongly typed with value and reference types:

- **Value Types**: `int`, `double`, `bool`, `struct`, `enum`.

- **Reference Types**: `string`, `object`, classes, arrays, delegates.
- **Special**: `dynamic`, `var` (type inference), `nint` (native-sized int).

```
int x = 42;
double pi = 3.14;
string name = "Alice";
bool active = true;
var inferred = 100; // int
dynamic dyn = "Can change type";
int? nullable = null; // Nullable<int>

Console.WriteLine($"x: {x}, name: {name}, nullable: {nullable}");
```

Nullable Reference Types (8.0+):

```
string? maybeNull = null;
string nonNull = "Must initialize";
Console.WriteLine(maybeNull?.Length ?? 0); // Safe navigation
```

Edge Cases:

- **Overflow**: `checked { int max = int.MaxValue + 1; }` throws `OverflowException`.
- **Null Dereference**: Enabled nullable reference types prevent at compile time.
- **Dynamic Overhead**: `dynamic` bypasses type checking, slower.

Cross-Connections:

- **Generics**: Type-safe collections.
- **LINQ**: Query typed data.
- **Records**: Immutable data types.

Functions and Methods

Methods are defined in classes; functions can be local or top-level (9.0+).

```
int Add(int a, int b = 0) => a + b;

class Calculator
{
    public static double Multiply(double x, double y) => x * y;

    public string Greet(string name, string greeting = "Hello")
    {
        return $"{greeting}, {name}!";
    }
}

// Local function
string FormatResult(int x)
{
    string Prefix() => x > 0 ? "Positive" : "Non-positive";
    return $"{Prefix()} {x}";
}
```

```
Console.WriteLine(Add(5, 3)); // 8
Console.WriteLine(Calculator.Multiply(2.5, 4)); // 10
Console.WriteLine(new Calculator().Greet("Alice")); // Hello, Alice!
Console.WriteLine(FormatResult(10)); // Positive 10
```

Advanced Features:

- **Expression-Bodied**: `=>` for concise methods.
- **Params Collections**: (13.0+) `void Log(params ReadOnlySpan<string> messages)`.
- **Ref/Out Parameters**: `ref int x`, `out int y`.

```
void Swap(ref int a, ref int b) => (a, b) = (b, a);
int TryParse(string s, out int result) => int.TryParse(s, out result);

int x = 1, y = 2;
Swap(ref x, ref y);
Console.WriteLine($"x: {x}, y: {y}"); // x: 2, y: 1
```

Edge Cases:

- **Default Parameters**: Evaluated at call site, not compile time.
- **Ref Safety**: `ref` locals must be initialized.
- **Overloading**: Ambiguity resolved by best match.

Cross-Connections:

- **Delegates**: Methods as first-class objects.
- **Async**: Methods with `async Task`.
- **LINQ**: Methods in query expressions.

Control Flow

C# supports `if`, `switch`, `for`, `foreach`, `while`, and pattern matching.

```
int x = 7;
if (x > 5)
    Console.WriteLine("Greater");
else if (x is 5)
    Console.WriteLine("Equal");
else
    Console.WriteLine("Lesser");

// Switch expression (8.0+)
string Describe(int n) => n switch
{
    > 0 => "Positive",
    0 => "Zero",
    < 0 => "Negative",
    _ => throw new ArgumentException("Invalid")
};
```

```
// Loops
for (int i = 1; i <= 3; i++)
    Console.WriteLine(i);

var numbers = new[] { 1, 2, 3 };
foreach (var n in numbers)
    Console.WriteLine(n);

// Pattern matching
object obj = "Hello";
if (obj is string { Length: > 3 } s)
    Console.WriteLine(s.ToUpper());
```

Advanced Features:

- **Switch Expressions**: Concise, exhaustive (8.0+).
- **Pattern Matching**: `is`, `switch` with types and properties.
- **Using Declarations**: Auto-dispose resources.

Edge Cases:

- **Switch Exhaustiveness**: Non-exhaustive cases throw at runtime.
- **Foreach Mutability**: Modifying collections during iteration throws.
- **Short-Circuiting**: `&&` and `||` avoid unnecessary evaluation.

Cross-Connections:

- **Pattern Matching**: Enhances control flow.
- **LINQ**: `foreach` over query results.
- **Error Handling**: Combine with `try`/`catch`.

Object-Oriented Programming

Classes and Objects

Classes define objects with fields, properties, and methods.

```
public class Person
{
    public string Name { get; init; } // Init-only (9.0+)
    private int age;

    public Person(string name, int age)
    {
        Name = name;
        this.age = age;
    }

    public string Introduce() => $"I'm {Name}, {age} years old";
}
```

```
var p = new Person("Alice", 30);
Console.WriteLine(p.Introduce()); // I'm Alice, 30 years old
```

Primary Constructors (12.0+):

```
public class Circle(double radius)
{
    public double Area => Math.PI * radius * radius;
}

var c = new Circle(5);
Console.WriteLine(c.Area); // 78.53981633974483
```

Edge Cases:

- **Field vs. Property**: Fields lack encapsulation; prefer properties.
- **Init-Only**: Immutable after construction.
- **Object Initializers**: `new Person { Name = "Bob" }` bypasses constructor.

Cross-Connections:

- **Records**: Immutable classes.
- **Interfaces**: Define contracts.
- **Generics**: Type-safe classes.

Inheritance and Interfaces

Inheritance extends classes; interfaces define contracts.

```
public interface IAnimal
{
    string Speak();
}

public abstract class Animal
{
    protected string Name { get; }

    protected Animal(string name) => Name = name;
}

public class Dog : Animal, IAnimal
{
    public Dog(string name) : base(name) { }

    public string Speak() => $"{Name} says Woof!";
}

var dog = new Dog("Buddy");
Console.WriteLine(dog.Speak()); // Buddy says Woof!
```

Advanced Features:

- **Sealed Classes/Methods**: Prevent further inheritance.

- **Default Interface Methods**: (8.0+) Provide implementation.
- **Record Inheritance**: (9.0+) Records can inherit records.

Edge Cases:

- **Multiple Inheritance**: Classes inherit one base; interfaces allow multiple.
- **Virtual vs. Override**: Incorrect overrides break polymorphism.
- **Interface Conflicts**: Explicit implementation resolves ambiguity.

Cross-Connections:

- **Generics**: Constrain to interfaces.
- **Testing**: Mock interfaces.
- **Pattern Matching**: Check interface types.

Records

Records (9.0+) provide immutable data types.

```
public record Person(string Name, int Age);

var p1 = new Person("Alice", 30);
var p2 = new Person("Alice", 30);
Console.WriteLine(p1 == p2); // true (value equality)
Console.WriteLine(p1 with { Age = 31 }); // Non-destructive mutation
```

Positional Records (12.0+):

```
public record Circle(double Radius)
{
    public double Area => Math.PI * Radius * Radius;
}
```

Edge Cases:

- **Equality**: Records compare properties, not references.
- **Inheritance**: Records can’t inherit classes, only records.
- **Serialization**: Records serialize as JSON objects.

Cross-Connections:

- **Immutability**: Aligns with functional programming.
- **LINQ**: Records in query results.
- **ASP.NET**: Records as DTOs.

Generics

Generics enable type-safe, reusable code.

```
public class Repository<T> where T : class
{
    private readonly List<T> items = [];

    public void Add(T item) => items.Add(item);
    public IEnumerable<T> GetAll() => items;
}

var repo = new Repository<string>();
repo.Add("Hello");
foreach (var item in repo.GetAll())
    Console.WriteLine(item); // Hello
```

Advanced Features:

- **Constraints**: `where T : struct`, `new()`, `IInterface`.
- **Covariance/Contravariance**: `out T`, `in T`.
- **Generic Methods**: `T Parse<T>(string s)`.

Edge Cases:

- **Type Erasure**: Generic types resolved at compile time.
- **Constraint Conflicts**: `where T : class, struct` is invalid.
- **Performance**: Generics avoid boxing for value types.

Cross-Connections:

- **Collections**: Built-in generics (`List<T>`).
- **LINQ**: Generic query methods.
- **Dependency Injection**: Generic services.

Delegates and Events

Delegates are type-safe function pointers; events enable pub-sub.

```
public delegate void Notify(string message);

public class Publisher
{
    public event Notify? OnMessage;

    public void Send(string message) => OnMessage?.Invoke(message);
}

var pub = new Publisher();
pub.OnMessage += m => Console.WriteLine(m);
pub.Send("Hello!"); // Hello!
```

Advanced Features:

- **Func/Action**: Built-in delegates: `Func<int, string>`, `Action<string>`.

- **Lambda Expressions**: `x => x * 2`.
- **Multicast Delegates**: Multiple subscribers.

Edge Cases:

- **Null Events**: Check with `?.Invoke`.
- **Memory Leaks**: Unsubscribe events to avoid references.
- **Thread Safety**: Events in multithreaded apps need locks.

Cross-Connections:

- **Async**: Delegates with `Task`.
- **LINQ**: Delegates in query expressions.
- **Testing**: Mock event handlers.

LINQ (Language Integrated Query)

LINQ queries collections with SQL-like syntax.

```
var numbers = new[] { 1, 2, 3, 4, 5 };
var evens = from n in numbers
            where n % 2 == 0
            orderby n descending
            select n * 2;

foreach (var n in evens)
    Console.WriteLine(n); // 8, 4

// Method syntax
var squares = numbers.Where(n => n % 2 == 0)
                     .OrderByDescending(n => n)
                     .Select(n => n * 2);
```

Advanced Features:

- **Deferred Execution**: Queries execute when enumerated.
- **Join/GroupBy**: Combine and aggregate data.
- **Queryable**: LINQ to SQL/EF for databases.

```
var people = new[] { new { Name = "Alice", Age = 30 }, new { Name = "Bob", Age = 25 }
};
var grouped = from p in people
              group p by p.Age / 10 into g
              select new { Decade = g.Key * 10, Count = g.Count() };

foreach (var g in grouped)
    Console.WriteLine($"Decade {g.Decade}: {g.Count}");
```

Edge Cases:

- **Deferred Execution Pitfalls**: Modifying source after query definition.

- **Performance**: Avoid complex LINQ in hot paths.
- **Null Handling**: `Where` skips nulls; `Select` may throw.

Cross-Connections:

- **Collections**: LINQ operates on `IEnumerable<T>`.
- **EF Core**: LINQ translates to SQL.
- **Testing**: Mock LINQ results.

Error Handling

C# uses `try`/`catch` for exceptions.

```
try
{
    int Divide(int a, int b) => a / b;
    Console.WriteLine(Divide(10, 0));
}
catch (DivideByZeroException ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}
finally
{
    Console.WriteLine("Cleanup");
}
```

Advanced Features:

- **Custom Exceptions**: Inherit from `Exception`.
- **Exception Filters**: `catch (Exception ex) when (ex.Message.Contains("specific"))`.
- **Throw Expressions**: `x ?? throw new ArgumentNullException()`.

```
public class CustomException : Exception
{
    public CustomException(string message, int code) : base(message) => Code = code;
    public int Code { get; }
}

try
{
    throw new CustomException("Failed", 400);
}
catch (CustomException ex) when (ex.Code == 400)
{
    Console.WriteLine($"Error {ex.Code}: {ex.Message}");
}
```

Edge Cases:

- **Swallowed Exceptions**: Avoid empty `catch` blocks.
- **Async Exceptions**: Use `await` to catch `Task` errors.

- **Performance**: Exceptions in loops are costly.

Cross-Connections:

- **Async**: Handle exceptions in `Task`.
- **Testing**: Assert exceptions.
- **ASP.NET**: Global exception handling.

Asynchronous Programming

Async/await enables non-blocking code.

```
public async Task<string> FetchAsync(string url)
{
    using var client = new HttpClient();
    return await client.GetStringAsync(url);
}

async Task Main()
{
    try
    {
        var content = await FetchAsync("https://api.example.com");
        Console.WriteLine(content);
    }
    catch (HttpRequestException ex)
    {
        Console.WriteLine($"Error: {ex.Message}");
    }
}

await Main();
```

Advanced Features:

- **Task.WhenAll**: Parallel tasks.
- **ValueTask**: Optimize small async operations.
- **IAsyncEnumerable**: Stream data (8.0+).

```
public async IAsyncEnumerable<int> GenerateAsync()
{
    for (int i = 1; i <= 3; i++)
    {
        await Task.Delay(100);
        yield return i;
    }
}

await foreach (var n in GenerateAsync())
    Console.WriteLine(n); // 1, 2, 3
```

Edge Cases:

- **Deadlocks**: Avoid `Task.Result` in async contexts.
- **Cancellation**: Use `CancellationToken`.
- **Resource Leaks**: Dispose async resources with `using`.

Cross-Connections:

- **LINQ**: Async queries with `IAsyncEnumerable`.
- **ASP.NET**: Async controllers.
- **Testing**: Mock async methods.

Collections and Data Structures

C# provides rich collections in `System.Collections.Generic`.

```
var list = new List<int> { 1, 2, 3 };
list.Add(4);
Console.WriteLine(string.Join(", ", list)); // 1, 2, 3, 4

var dict = new Dictionary<string, int> { ["a"] = 1, ["b"] = 2 };
Console.WriteLine(dict["a"]); // 1

var set = new HashSet<int> { 1, 2, 2 };
Console.WriteLine(set.Count); // 2
```

Collection Expressions (12.0+):

```
int[] arr = [1, 2, 3];
List<int> spread = [..arr, 4]; // [1, 2, 3, 4]
```

Advanced Features:

- **Concurrent Collections**: `ConcurrentDictionary`, `BlockingCollection`.
- **Immutable Collections**: `ImmutableList<T>` (NuGet: `System.Collections.Immutable`).
- **Span<T>/Memory<T>**: High-performance memory slices.

```
Span<int> span = stackalloc int[] { 1, 2, 3 };
Console.WriteLine(span[1]); // 2
```

Edge Cases:

- **Thread Safety**: Non-concurrent collections require locks.
- **Memory Usage**: Large collections impact GC.
- **Null Keys**: `Dictionary` throws on null keys.

Cross-Connections:

- **LINQ**: Query collections.
- **Generics**: Type-safe collections.
- **Performance**: Optimize with `Span<T>`.

Advanced Topics

Pattern Matching

Pattern matching (7.0+) enhances type and value checks.

```
object obj = "Hello";
string result = obj switch
{
    string { Length: > 3 } s => s.ToUpper(),
    int n when n > 0 => $"Positive {n}",
    _ => "Unknown"
};

Console.WriteLine(result); // HELLO
```

Advanced Features:

- **Type Patterns**: `is T x`.
- **Property Patterns**: `is { Prop: var x }`.
- **List Patterns**: (11.0+) `[1, .., n]`.

Edge Cases:

- **Null Handling**: Patterns handle `null` explicitly.
- **Exhaustiveness**: Non-exhaustive `switch` may throw.
- **Performance**: Complex patterns add overhead.

Cross-Connections:

- **Control Flow**: Replaces `if`/`switch`.
- **Records**: Ideal for matching immutable data.
- **LINQ**: Combine with queries.

Source Generators

Source generators (8.0+) generate code at compile time.

```
// MyGenerator.cs
[Generator]
public class HelloGenerator : ISourceGenerator
{
    public void Initialize(GeneratorInitializationContext context) { }

    public void Execute(GeneratorExecutionContext context)
    {
        context.AddSource("Generated.cs", """
            namespace Generated;
            public static class Hello
            {
                public static void Say() => Console.WriteLine("Generated!");
            }
```

```
            """);
    }
}
```

Usage:

```
Generated.Hello.Say(); // Generated!
```

Edge Cases:

- **Debugging**: Generated code harder to trace.
- **Build Dependency**: Generators run during compilation.
- **Performance**: Minimize generation time.

Cross-Connections:

- **Attributes**: Drive generator logic.
- **Reflection**: Alternative at runtime.
- **Testing**: Verify generated code.

Native Interop

Call native code via P/Invoke or COM.

```
using System.Runtime.InteropServices;

public static class Native
{
    [DllImport("mylib.dll")]
    public static extern int Double(int x);
}

Console.WriteLine(Native.Double(5)); // 10
```

C Code (`mylib.c`):

```
int Double(int x) { return x * 2; }
```

Compile (Windows):

```
gcc -shared -o mylib.dll mylib.c
```

Advanced Features:

- **StructLayout**: Control memory layout.
- **LibraryImport**: (7.0+) Safer P/Invoke.
- **COM Interop**: Access Windows COM objects.

Edge Cases:

- **Platform Dependency**: DLLs differ by OS.
- **Memory Safety**: Incorrect pointers cause crashes.

- **Performance**: Marshaling overhead.

Cross-Connections:

- **Performance**: Optimize bottlenecks.
- **Memory Management**: Manage native resources.
- **ASP.NET**: Rare, but used for legacy integration.

Memory Management

C# uses garbage collection (GC) with manual options.

```
public class Resource : IDisposable
{
    private bool disposed;
    public void Dispose()
    {
        if (!disposed)
        {
            // Release resources
            disposed = true;
        }
        GC.SuppressFinalize(this);
    }
    ~Resource() => Dispose();
}

using var res = new Resource(); // Auto-dispose
```

Advanced Features:

- **Span<T>/Memory<T>**: Stack-based memory.
- **Unsafe Code**: `unsafe { int* p = stackalloc int[10]; }`.
- **GC Tuning**: `GC.Collect`, `GCSettings.LargeObjectHeapThreshold`.

Edge Cases:

- **Memory Leaks**: Event handlers, static fields.
- **Finalizers**: Non-deterministic, avoid heavy logic.
- **Pinned Objects**: Prevent GC movement for P/Invoke.

Cross-Connections:

- **Native Interop**: Manage native memory.
- **Performance**: Minimize GC pressure.
- **Collections**: Optimize allocations.

Reflection and Attributes

Reflection inspects types; attributes add metadata.

```
[AttributeUsage(AttributeTargets.Method)]
public class LogAttribute : Attribute { }

public class Service
{
    [Log]
    public void Execute() => Console.WriteLine("Running");
}

var method = typeof(Service).GetMethod("Execute");
if (method.GetCustomAttribute<LogAttribute>() != null)
    Console.WriteLine("Method is logged");
```

Advanced Features:

- **Dynamic Types**: `TypeBuilder` for runtime types.
- **Expression Trees**: Compile dynamic code.
- **Custom Attributes**: Drive behavior (e.g., ASP.NET routing).

Edge Cases:

- **Performance**: Reflection is slow; cache results.
- **Security**: Reflection can bypass access modifiers.
- **AOT Compilation**: Reflection may fail in trimmed apps.

Cross-Connections:

- **Source Generators**: Replace runtime reflection.
- **ASP.NET**: Attributes for routing/validation.
- **Testing**: Inspect types in tests.

Performance Optimization

Optimize C# with profiling and techniques.

```
// Use StringBuilder for concatenation
var sb = new StringBuilder();
for (int i = 0; i < 1000; i++)
    sb.Append(i);
Console.WriteLine(sb.ToString());
```

Tools:

- **BenchmarkDotNet**: Microbenchmarking.
- **dotTrace**: JetBrains profiler.
- **Visual Studio Profiler**: Built-in diagnostics.

Advanced Features:

- **AOT Compilation**: (7.0+) Native binaries for startup speed.

- **Span<T>**: Reduce allocations.
- **ValueTask**: Avoid `Task` allocations.

Edge Cases:

- **Micro-Optimizations**: Premature optimization wastes effort.
- **GC Pressure**: Large objects trigger frequent collections.
- **Inlining**: JIT may skip small methods.

Cross-Connections:

- **Memory Management**: Optimize allocations.
- **Async**: Minimize await overhead.
- **LINQ**: Avoid complex queries in hot paths.

Security Best Practices

Secure C# applications against vulnerabilities.

```
// Prevent SQL injection with EF Core
using var db = new AppDbContext();
var user = await db.Users
    .FirstOrDefaultAsync(u => u.Name == name); // Parameterized

// XSS prevention in ASP.NET
var encoded = System.Web.HttpUtility.HtmlEncode(userInput);

// Secure password hashing
string hash = BCrypt.Net.BCrypt.HashPassword(password);
bool valid = BCrypt.Net.BCrypt.Verify(password, hash);
```

Advanced Practices:

- **Dependency Scanning**: Use `dotnet list package --vulnerable`.
- **Secure APIs**: Use `[Authorize]` in ASP.NET.
- **Data Protection**: `IDataProtectionProvider` for encryption.

Edge Cases:

- **Injection**: Always parameterize inputs.
- **Secrets**: Avoid hardcoding; use Azure Key Vault or environment variables.
- **CORS**: Restrict origins in ASP.NET.

Cross-Connections:

- **ASP.NET**: Built-in security features.
- **EF Core**: Parameterized queries.
- **Testing**: Verify security controls.

Testing in C#

xUnit

xUnit is a popular testing framework.

```
// Tests/MathTests.cs
public class MathTests
{
    [Fact]
    public void Add_ReturnsSum()
    {
        Assert.Equal(5, Add(2, 3));
        Assert.NotEqual(6, Add(2, 3));
    }

    [Theory]
    [InlineData(1, 2, 3)]
    [InlineData(0, 0, 0)]
    public void Add_Parameterized(int a, int b, int expected)
    {
        Assert.Equal(expected, Add(a, b));
    }

    private int Add(int a, int b) => a + b;
}
```

Run:

```
dotnet test
```

Moq

Mock dependencies with Moq.

```
// Tests/ServiceTests.cs
public class ServiceTests
{
    [Fact]
    public void GetUser_ReturnsUser()
    {
        var repoMock = new Mock<IRepository>();
        repoMock.Setup(r => r.Find(1)).Returns(new User { Name = "Alice" });

        var service = new UserService(repoMock.Object);
        var user = service.GetUser(1);

        Assert.Equal("Alice", user.Name);
        repoMock.Verify(r => r.Find(1), Times.Once());
    }
}

public interface IRepository { User Find(int id); }
public class UserService
{
    private readonly IRepository repo;
```

```
    public UserService(IRepository repo) => this.repo = repo;
    public User GetUser(int id) => repo.Find(id);
}
```

Code Coverage

Measure coverage with `coverlet`:

```
dotnet add package coverlet.collector
dotnet test --collect:"XPlat Code Coverage"
```

Advanced Features:

- **Data-Driven Tests**: `[Theory]` with `[InlineData]`.
- **Mocking Async**: `Setup(r => r.FindAsync(It.IsAny<int>())).ReturnsAsync(...)`.
- **Integration Tests**: Test EF Core or APIs.

Edge Cases:

- **Over-Mocking**: Avoid mocking internal logic.
- **Test Pollution**: Isolate test state.
- **Coverage Gaps**: High coverage ≠ bug-free.

Cross-Connections:

- **Dependency Injection**: Mock services.
- **ASP.NET**: Test controllers.
- **EF Core**: Use in-memory database for tests.

.NET Ecosystem

ASP.NET Core

Build web apps and APIs.

```
// Program.cs
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
var app = builder.Build();
app.MapControllers();
app.Run();

// Controllers/UserController.cs
[ApiController]
[Route("api/[controller]")]
public class UserController : ControllerBase
{
    [HttpGet("{name}")]
    public IActionResult Greet(string name) => Ok(new { Message = $"Hello, {name}!"
});
}
```

Entity Framework Core

ORM for database access.

```
// Models/User.cs
public class User
{
    public int Id { get; set; }
    public string Name { get; set; } = null!;
}

// Data/AppDbContext.cs
public class AppDbContext : DbContext
{
    public DbSet<User> Users { get; set; }
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }
}

// Program.cs
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlite("Data Source=app.db"));

using (var scope = app.Services.CreateScope())
{
    var db = scope.ServiceProvider.GetRequiredService<AppDbContext>();
    db.Database.EnsureCreated();
}
```

Blazor

Build web UIs with C#.

```
// Pages/Greeting.razor
@page "/greet"
<h3>Enter Name</h3>
<input @bind="name" />
<p>Hello, @name!</p>

@code {
    private string name = "Guest";
}
```

Advanced Features:

- **Dependency Injection**: Built-in IoC container.
- **Middleware**: ASP.NET request pipeline.
- **Change Tracking**: EF Core optimizes updates.

Edge Cases:

- **N+1 Queries**: Eager-load relations in EF Core.
- **Routing Conflicts**: Unique routes in ASP.NET.
- **Blazor Performance**: Minimize re-renders.

Cross-Connections:

- **LINQ**: EF Core queries.
- **Async**: Pervasive in ASP.NET/EF.
- **Testing**: Test APIs and components.

Building a Sample Project

This project implements a **task management API** using ASP.NET Core, EF Core, async processing, and native interop, showcasing C#'s modern features.

Project Structure:

```
TaskManager/
├── TaskManager/
│   ├── Models/
│   │   └── Task.cs
│   ├── Services/
│   │   └── TaskService.cs
│   ├── Data/
│   │   └── AppDbContext.cs
│   ├── Controllers/
│   │   └── TasksController.cs
│   ├── Native/
│   │   └── mylib.c
│   ├── Program.cs
│   └── TaskManager.csproj
├── TaskManager.Tests/
│   ├── Unit/
│   │   └── TaskTests.cs
│   ├── Integration/
│   │   └── ApiTests.cs
│   └── TaskManager.Tests.csproj
└── TaskManager.sln
```

TaskManager.csproj:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="9.0.0"
/>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="9.0.0"
/>
    <PackageReference Include="System.Runtime.InteropServices" Version="9.0.0" />
  </ItemGroup>
</Project>
```

TaskManager.Tests.csproj:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="xunit" Version="2.9.2" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.9.2" />
    <PackageReference Include="Moq" Version="4.20.70" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.11.0" />
    <PackageReference Include="coverlet.collector" Version="6.0.2" />
  </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\TaskManager\TaskManager.csproj" />
  </ItemGroup>
</Project>
```

Native/mylib.c:

```
int ComputePriority(int id) {
    return id * 3;
}
```

Compile (Windows):

```
gcc -shared -o TaskManager/bin/Debug/net9.0/mylib.dll Native/mylib.c
```

Models/Task.cs:

```
public enum TaskStatus { Pending, Completed, Failed }

public record Task(int Id, string Description, TaskStatus Status =
TaskStatus.Pending, int Priority = 0)
{
    public DateTime CreatedAt { get; init; } = DateTime.UtcNow;
    public DateTime? UpdatedAt { get; set; }
}
```

Data/AppDbContext.cs:

```
using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext
{
    public DbSet<Task> Tasks { get; set; } = null!;

    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Task>()
            .Property(t => t.Status)
            .HasConversion<string>();
    }
}
```

Services/TaskService.cs:

```
using System.Runtime.InteropServices;

public class TaskService
{
    private readonly AppDbContext db;
    private readonly IHostApplicationLifetime lifetime;

    public TaskService(AppDbContext db, IHostApplicationLifetime lifetime)
    {
        this.db = db;
        this.lifetime = lifetime;
        StartAsyncProcessor();
    }

    [DllImport("mylib.dll")]
    private static extern int ComputePriority(int id);

    public async Task<Task> CreateTaskAsync(Task task, CancellationToken ct =
default)
    {
        task = task with { Priority = ComputePriority(task.Id) };
        db.Tasks.Add(task);
        await db.SaveChangesAsync(ct);
        return task;
    }

    public async Task<IEnumerable<Task>> GetTasksAsync(CancellationToken ct =
default)
        => await db.Tasks.ToListAsync(ct);

    private void StartAsyncProcessor()
    {
        Task.Run(async () =>
        {
            while (!lifetime.ApplicationStopping.IsCancellationRequested)
            {
                var pending = await db.Tasks
                    .Where(t => t.Status == TaskStatus.Pending)
                    .FirstOrDefaultAsync();
                if (pending != null)
                {
                    pending = pending with
                    {
                        Status = TaskStatus.Completed,
                        UpdatedAt = DateTime.UtcNow
                    };
                    db.Tasks.Update(pending);
                    await db.SaveChangesAsync();
                }
                await Task.Delay(1000);
            }
        });
    }
}
```

Controllers/TasksController.cs:

```
using Microsoft.AspNetCore.Mvc;

[ApiController]
[Route("api/[controller]")]
public class TasksController : ControllerBase
{
    private readonly TaskService service;

    public TasksController(TaskService service) => this.service = service;

    [HttpPost]
    public async Task<IActionResult> Create([FromBody] Task task)
    {
        var created = await service.CreateTaskAsync(task);
        return CreatedAtAction(nameof(Get), new { id = created.Id }, created);
    }

    [HttpGet]
    public async Task<IEnumerable<Task>> Get() => await service.GetTasksAsync();

    [HttpGet("{id}")]
    public async Task<IActionResult> Get(int id)
    {
        var task = await service.GetTasksAsync()
            .ContinueWith(t => t.Result.FirstOrDefault(t => t.Id == id));
        return task != null ? Ok(task) : NotFound();
    }
}
```

Program.cs:

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlite("Data Source=tasks.db"));
builder.Services.AddScoped<TaskService>();

var app = builder.Build();
app.MapControllers();

using (var scope = app.Services.CreateScope())
{
    var db = scope.ServiceProvider.GetRequiredService<AppDbContext>();
    await db.Database.EnsureCreatedAsync();
}

await app.RunAsync();
```

Tests/Unit/TaskTests.cs:

```
using Xunit;
```

```
public class TaskTests
{
    [Fact]
    public void Task_Equality()
    {
        var t1 = new Task(1, "Test", TaskStatus.Pending, 5);
        var t2 = new Task(1, "Test", TaskStatus.Pending, 5);
        Assert.Equal(t1, t2);
    }

    [Theory]
    [InlineData(1, "Test", TaskStatus.Pending)]
    public void Task_Creation(int id, string desc, TaskStatus status)
    {
        var task = new Task(id, desc, status);
        Assert.Equal(id, task.Id);
        Assert.Equal(desc, task.Description);
        Assert.Equal(status, task.Status);
    }
}
```

Tests/Integration/ApiTests.cs:

```
using Microsoft.AspNetCore.Mvc.Testing;
using System.Net.Http.Json;
using Xunit;

public class ApiTests : IClassFixture<WebApplicationFactory<Program>>
{
    private readonly HttpClient client;

    public ApiTests(WebApplicationFactory<Program> factory)
    {
        client = factory.CreateClient();
    }

    [Fact]
    public async Task CreateTask_ReturnsCreated()
    {
        var task = new Task(1, "Test Task");
        var response = await client.PostAsJsonAsync("/api/tasks", task);

        response.EnsureSuccessStatusCode();
        var created = await response.Content.ReadFromJsonAsync<Task>();

        Assert.Equal(1, created?.Id);
        Assert.Equal("Test Task", created?.Description);
    }
}
```

Features Demonstrated:

- **OOP**: Records, classes, interfaces.
- **Async**: Async methods, background processing.

- **Native Interop**: P/Invoke for priority computation.
- **Database**: EF Core with SQLite.
- **Web Development**: ASP.NET Core REST API.
- **Testing**: xUnit for unit and integration tests.
- **Type Safety**: Nullable reference types, records.
- **Security**: Parameterized queries via EF Core.
- **Dependency Injection**: Scoped services.

Running the Project:

```
dotnet new sln -n TaskManager
cd TaskManager
dotnet sln add TaskManager
dotnet sln add TaskManager.Tests
gcc -shared -o TaskManager/bin/Debug/net9.0/mylib.dll Native/mylib.c # Windows
# For Linux: gcc -shared -o TaskManager/bin/Debug/net9.0/libmylib.so Native/mylib.c
dotnet build
dotnet run --project TaskManager
```

Test:

```
dotnet test
```

Sample API Usage:

```
curl -X POST http://localhost:5000/api/tasks -H "Content-Type: application/json" -d
'{"Id":1,"Description":"Do homework"}'
curl http://localhost:5000/api/tasks
```

Edge Cases Handled:

- **Error Handling**: EF Core exceptions, HTTP status codes.
- **Async Safety**: Cancellation tokens, background tasks.
- **Security**: No injection risks via EF Core.
- **Type Safety**: Records and nullable types.
- **Platform Portability**: SQLite for cross-platform; native DLL handling.

Resources

- **Official Docs**:
  - C# Guide,
  - .NET Docs,
  - ASP.NET Core Docs,
  - EF Core Docs.

- **Tutorials**:
  - Microsoft Learn,
  - Pluralsight C#,
  - C# Station.
- **Community**:
  - Stack Overflow,
  - Reddit r/csharp,
  - .NET Community.
- **Tools**:
  - Visual Studio,
  - VS Code,
  - Rider,
  - BenchmarkDotNet.
- **Libraries**:
  - NuGet,
  - Awesome .NET.
- **Books**:
  - *C# 12 and .NET 9 in Action* (Manning),
  - *Pro C# 12 with .NET 9* (Apress),
  - *CLR via C#* (Microsoft Press).
- **Security**:
  - OWASP .NET,
  - Microsoft Security.

This guide and sample project provide a comprehensive foundation for mastering C#, from basic syntax to advanced .NET development, with practical applications and deep insights into its versatile features.