CS321

Advanced Programming Techniques, Bishop's University, Winter 2023
https://github.com/cdiggins/cs321

About

This class is a follow-up to CS211, Introduction to Programming.

We will be using C# 7 as the language of instruction.

Emphasis on hands-on assignments culminating in a final project.

Goals

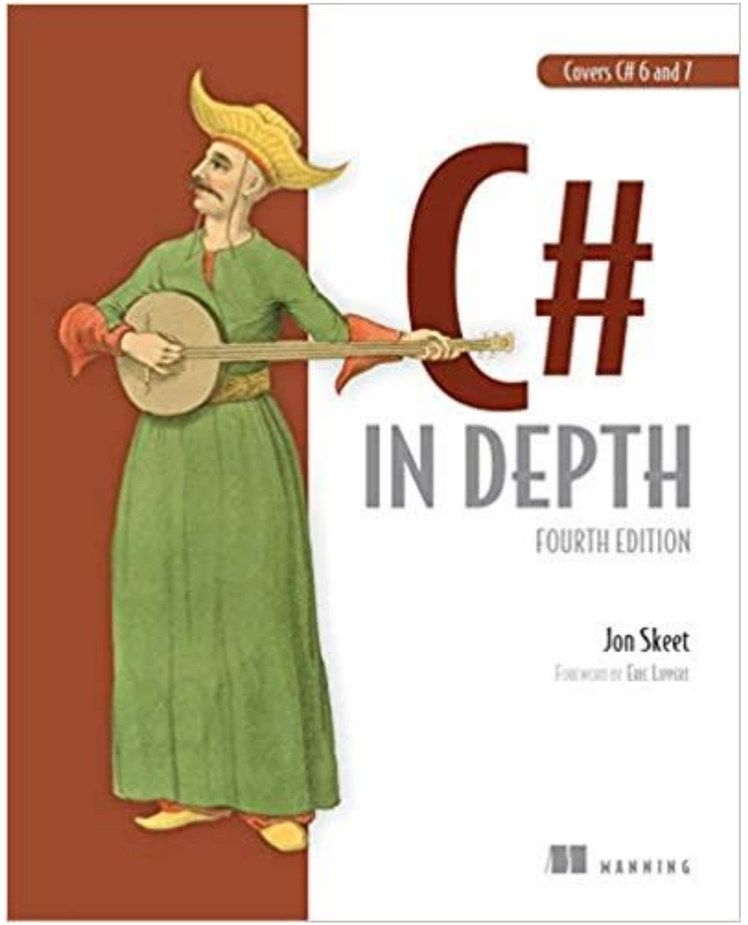
Introduce you to the craft of software development

Deepen your knowledge of various programming approaches

Build a solid proficiency in reading and writing C#

Reference Materials

- Official C# Language Reference: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference
- Supplemental reading:
  "C# In Depth Fourth Edition" by Jon Skeet
- Notes posted to GitHub at:
  https://github.com/cdiggins/cs321



Course Evaluation

20% = 5 Assignments

25% = 8 Labs

15% = Midterm (closed book)

40% = Final Project (individual)

Course Progression and Division

- Labs focus on smaller tasks and tools
- Assignments focus on concepts, design, ideas, planning
- Best practices become more important later
- Midterm will be closed book
- Projects to be done individually
- Helping each other out is encouraged, be careful to do your own work
- Class will become more challenging towards the end

Plagiarism

Plagiarism is a kind of academic dishonesty in which an individual uses the work of another without appropriate acknowledgement.

Plagiarism includes but is not limited to the following practices:

- Using another's work without acknowledgement
- Copying material without quotation marks
- Paraphrasing too closely the exact words of the originating author
- Submitting work written in whole or in part as one's own by another individual.

Academic Integrity

- Using an AI tool for help is okay (like GitHub Copilot)
- Using StackOverflow.com is fine
- Asking friends for help is fine
- Plagiarism is *not acceptable!*
- Credit all of your sources: friends, internet, or AI
- Do the work yourself, use other sources as tools

Before and After Midterm

Part one focuses on:

- Become comfortable and confident with C#
- learning a range of techniques
- Improving programming skills
- Developing a solid foundation

Part two focuses on:

- Building larger pieces of software
- Advanced features of the C# language
- Putting complex things together
- Learning best practices and patterns

Course Outline – Part 1

About Class and Overview
Introduction to C#
System Library
Functions as Data
Introduction to Object Oriented Programming
Reflection and Introspection
Testing, Debugging, and Error Handling

Course Outline – Part 2

- Coding Style and Principles
- Advanced features of C#
- Advanced topics in Object Oriented Programming
- Concurrency and Asynchronous Programming
- Common Software Patterns
- GUI Programming
- Foreign Function Interfaces and Marshaling

About Midterm

Closed book

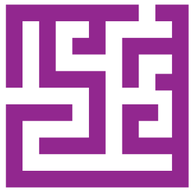
Verify a basic proficiency in C#

Assess understanding of key concepts

Covers content taught in Part 1

Covers material only in notes and authoritative source

Final Project



Put together a robust piece of non-trivial software

Details will be provided later in the semester

Leverages concepts learned in part 2

CRAFT OF SOFTWARE DEVELOPMENT

What is a Craft

From Wikipedia: A craft or trade is a pastime or an occupation that requires particular skills and knowledge of skilled work. In a historical sense, particularly the Middle Ages and earlier, the term is usually applied to people occupied in small scale production of goods, or their maintenance, for example by tinkers.

Seems to be a good definition!

Software Development is Exciting

We don’t know very much yet

Ideas and approaches are evolving rapidly

Continuously informed by mathematics and science

We will eventually be able to do practically anything

Four Steps to Software Development

UNDERSTAND

PLAN

IMPLEMENT

MEASURE

ABOUT C#

An overview of the language and run-time

About C#

C# is a mature and efficient general-purpose high-level language with good support for object-oriented and functional programming techniques.

C# programs are usually compiled to IL (intermediate language) byte-code and executed within the .NET execution environment.

https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

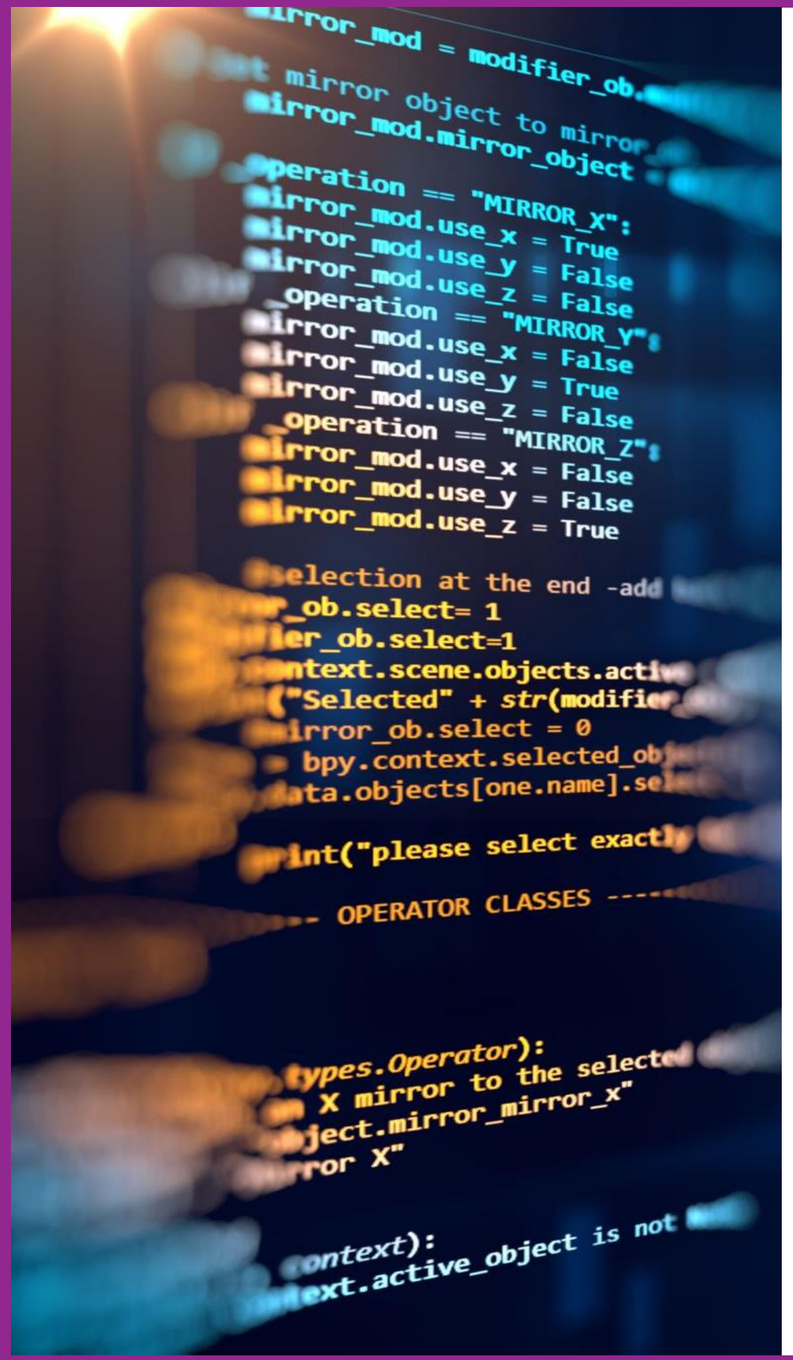
Some places where C# is Used

- Windows application programming
- Linux and Mac application development
- Mobile app development (Xamarin and Maui)
- Unity scripting
- Plug-in Development for Windows applications
- Server-Side scripting (ASP.NET)
- Limited usage Web Client (Blazor)

Assemblies and JIT Compilation

- Typically, .NET apps are compiled to intermediate language (IL) and is contained in assemblies.
- At run time, the just-in-time (JIT) compiler of the CLR (Common Language Runtime) translates IL to native code
- Assemblies take the form of executables (.exe) or dynamically linked libraries (.dll)
- An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality

https://learn.microsoft.com/en-us/dotnet/standard/assembly/



Ahead of Time (AOT) Compilation

.NET Native compiles UWP apps directly to native code.

- See : https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/

The Unity IL2CPP (Intermediate Language To C++) scripting backend converts IL code into C++ code, then uses the C++ code to create a native binary file.

- See: https://docs.unity3d.com/Manual/IL2CPP.html

Some Limitations of AOT Compilation

- No dynamic loading (for example, Assembly.LoadFile)
- No runtime code generation (for example, System.Reflection.Emit)
- No built-in COM (only applies to Windows)
- Some reflection APIs don’t work as expected
- Implies compilation into a single file
- Apps include required runtime libraries increasing their size
- Limited diagnostic support (debugging and profiling).

C# REPL

Tools exists for interpreting C# in a Read-Eval-Print Loop (REPL)

See the C# Interactive Window in Visual Studio for Windows

Or use the CSI.exe tool installed with Mono for Mac

Execution Environment

.NET provides a run-time environment called the common language runtime that runs the code and provides services that make the development process easier. Code that that targets the runtime is called managed code.

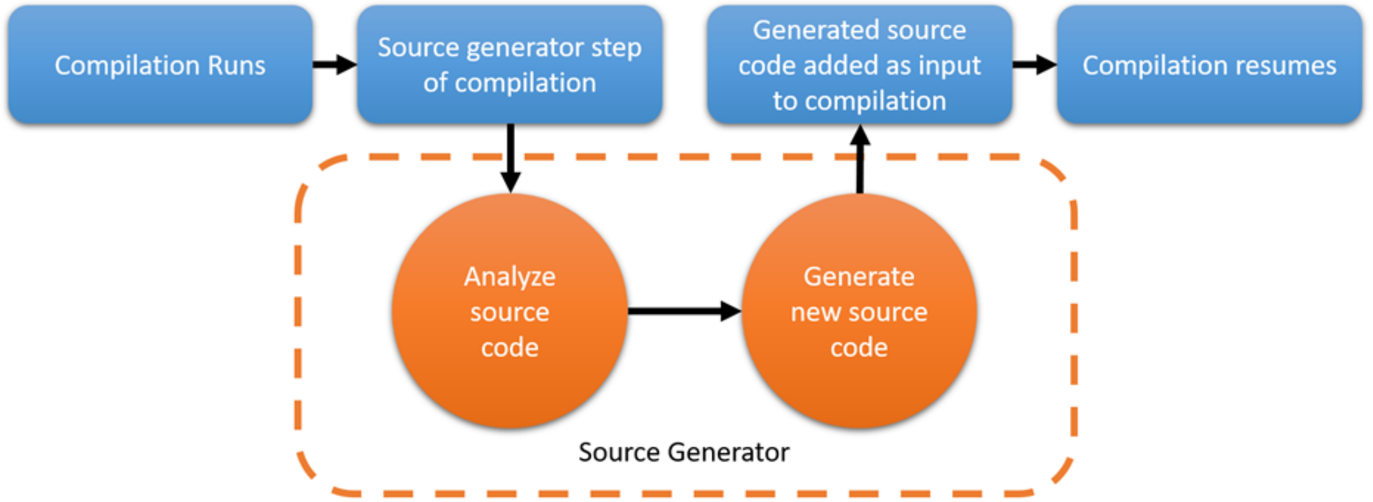
https://learn.microsoft.com/en-us/dotnet/standard/clr

Unsafe Code and Pointers

- By default C# generates code that is verifiable
- C# supports pointers within unsafe contexts
- Unsafe code can only be used within unsafe blocks within unsafe projects.
- Best to only use within projects designed for interfacing with low-level code.
- Not necessary for high-performance code.

Source Code Generation

- C# doesn't have a macro pre-processor but supports a few directives for conditional compilation
- Tool-chain supports compile-time source generators



C# Versions: a new one every 2-3 years

- C# 1.0 – 2002 a nice little language
- C# 2.0 – 2005 generics: now we are talking
- C# 3.0 – 2007 lambda: WOW!
- C# 4.0 – 2010 dynamic keyword
- C# 5.0 – 2012 async / await (important, but took awhile to Grok)
- C# 6.0 – 2015 lots of syntactic sugar
- C# 7.0 – 2017 tuples and deconstruction

Wait, what??

- C# 7.1 – 2017
- C# 7.2 – 2017
- C# 7.3 – 2018
- C# 8.0 – 2018
- C# 9.0 – 2020
- C# 10.0 – 2021
- C# 11.0 – 2022

.NET Runtime and Default C# Language

C# 11 is supported only on .NET 7 and newer versions. C# 10 is supported only on .NET 6 and newer versions. C# 9 is supported only on .NET 5 and newer versions.

Target	Version	C# language version default
.NET	7.x	C# 11
.NET	6.x	C# 10
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	all	C# 7.3

.NET Standard 2.0 (Most Portable)

Targets .NET Framework since 4.6.2 and Unity 2018.1, as well as other .NET implementations.

1.0 1.1 1.2 1.3 1.4 1.5 1.6 **2.0** 2.1

.NET Standard 2.0 has 32,638 of the 37,118 available APIs.

.NET implementation	Version support
.NET and .NET Core	2.0, 2.1, 2.2, 3.0, 3.1, 5.0, 6.0, 7.0
.NET Framework [1]	4.6.1 [2], 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
Mono	5.4, 6.4
Xamarin.iOS	10.14, 12.16
Xamarin.Mac	3.8, 5.16
Xamarin.Android	8.0, 10.0
Universal Windows Platform	10.0.16299, TBD
Unity	2018.1

Some C# and C++ Differences

- C# is garbage collected
- C# produced verifiable byte code which is compiled "just in time" (JIT)
- C# supports introspection, reflection, and run-time code generation
- In C# classes have different semantics than structs
- C# pointer types and operations are allowed, but only in *unsafe* contexts
- C# strings and arrays are classes
- C# supports only single inheritance, but has interfaces

Things I like about C# as a Language

Statically typed	Interfaces with extension methods	Safe and well-defined	Garbage collected
Good interoperability	High quality libraries	Excellent tooling	Good performance

ELEMENTS OF CODE

A refresher using C#

Key Elements of a Program

Data types

Functions

Statements

Expressions

Variables

Data Types

- A data type is a category of data values. As a programmer you can define new types using class, struct, interface, and enum declarations.

- Data type declarations define operations, member functions, and member variables.

```
// Represents a 2D pixel coordinate
public class Coordinate
{
    // constructor
    public Coordinate(int x, int y)
    {
        X = x;
        Y = y;
    }

    // member field
    public readonly int X;
    public readonly int Y;

    // member function
    public Coordinate Offset(int offsetX, int offsetY)
    {
        return new Coordinate(X + offsetX, Y + offsetY);
    }
}
```

Functions

- A function in programming is a code block that has zero or more input values (parameters), and an optional output.

```
int Quadratic(int a, int b, int c, int x)
{
    return (a * x * x) + (b * x) + c;
}
```

- Unlike functions in mathematics, they don't have to have a return value and may cause side-effects to happen

```
void WriteHello()
{
    Console.WriteLine("Hello");
}
```

Statements

- A statement is a unit of code that conceptually represents a single action. Statements may be compound, consisting of embedded statements, or are simple and terminated with the ';' character.
- Some common examples of simple statements include:
  - Return statement - return (a * x * x) + (b * x) + c;
  - Variable declaration - var a = 42;
  - Assignment statement - x = Quadratic(1,2,3,4);
  - Function call statement - Debug.Log("hello");

Literal Expressions

A literal expression is a representation of a value directly in code.

- Integer literal - `-3`, `42`, `0xFF`
- Floating point literal - `3.14f`, `2.13e+45`
- String literal - `“hello”`
- Boolean literal - `true` and `false`
- Character literal - `'a'`, `'\n'`, `'\u263A'`

Values

- A value is a piece of data such as a number or Boolean or object.
- Values are created by a computer program by evaluating expressions.
- Every value has a type.
- Values never change, but variables might change which value they refer to.
- Values are sometimes, but not always, stored in memory.
- Pretend they are, it makes it easier.

Types

A type is a category of values. C# comes with a few types built in (primitives).

- **int** – a whole number (integer) between approx. -2 billion and +2 billion
- **char** – a text character, such as 'a' or '9' or '/n'
- **string** – a sequence of zero or more text characters.
- **bool** – a Boolean logic value representing true or false
- **double** – a numerical value with a decimal point (e.g., 3.141)
- **byte** – an unsigned whole number between 0 and 255.

User Defined Types

- New types can be defined using class, struct, interface, and enum declarations.
- Types can be placed in a library to be reused.
- Types may contain:
  - Member variable (Fields)
  - Member functions (Method)
  - Operations
  - Constructors
- Types may have per-instance data, or static shared data.
- Types can inherit behavior or data from other types

Class

- The most common kind of user defined type is a class.
- A class is a set of data elements (fields) that can be treated as a single entity along with functions and operations for this type.
- Instances of a class are called objects.
- Class instances are created using the “new” keywords
- Class instances are initialized using the constructor special method.

```
// Represents a 2D pixel coordinate
public class Coordinate
{
    // constructor
    public Coordinate(int x, int y)
    {
        X = x;
        Y = y;
    }

    // member field
    public readonly int X;
    public readonly int Y;

    // member function
    public Coordinate Offset(int offsetX, int offsetY)
    {
        return new Coordinate(X + offsetX, Y + offsetY);
    }
}
```

Control Flow

The order in which the statements, functions, and expressions are executed or evaluated at run-time by a thread of execution.

Some statements affect control flow by choosing which statement is executed next, or the number of times a statement is executed

Some expressions also affect control flow (e.g., short-circuiting)

Common Statement Types

- **Branching statements** - branch statements affect control flow by determining what statement is executed next
- **Loop Statements** – loops affect control flow by executing an embedded statement, repeatedly while a condition evaluates to true.
- **Variable declarations** - declares a new variable and optionally initializes it
- **Expression statements** – either assign a value to a variable or execute a function
- **Block statements** - a set of zero more statements delimited by curly braces {} and that creates a new variable declaration space

C# LANGUAGE

High-Level Details

Organization of a C# Program

Assemblies

- Compiled from one or more source files
- May be a .exe or a .dll

Source files

- Using declarations
- Zero or one namespace

Namespaces

- Contain types
- May be split across files

Types

- Contain static and/or instance members
- Always in one namespace
- But may be split across files

Kinds of Types

- Classes
- Structs
- Interfaces
- Enums
- Delegates

Reference Types

- In C# classes are always allocated on the heap
- Unlike C++ their memory address can change.
- Means that in unsafe mode you have to “pin” pointers
- This prevents the GC from moving or reclaiming the memory
- Interfaces are also reference types
- All reference types inherit from System.Object

Value Types

- Most built-in types in C# are value types.
- Value types are defined in C# using the “struct” keyword
- This means that they are defined on the stack and have copy-by-value semantics
- Value types cannot inherit from other types

Built-in Types

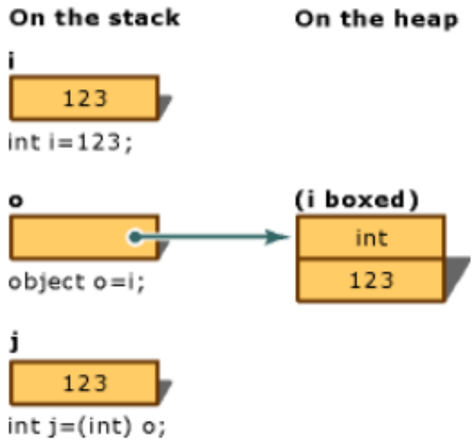
The following table lists the C# built-in reference types:

C# type keyword	.NET type
object	System.Object
string	System.String
dynamic	System.Object

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

bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
nint	System.IntPtr
nuint	System.UIntPtr
long	System.Int64
ulong	System.UInt64
short	System.Int16
ushort	System.UInt16

```
int i = 123;       // a value type
object o = i;      // boxing
int j = (int)o;    // unboxing
```



Boxing

- Value types can be implicitly cast into a System.Object reference type
- This process is called boxing
- Explicitly casting from a boxed value type to a value type is called unboxing
- https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/types/boxing-and-unboxing
- We will review this in practice