CSCI 1301 Book

https://csci-1301.github.io/about#authors

May 20, 2021 (10:32:42 PM)

Contents

1 Introduction to Computers and Programming

1.1 Principles of Computer Programming

- Computer hardware changes frequently - from room-filling machines with punch cards and tapes to modern laptops and tablets
- Capabilities of computers have changed rapidly (storage, speed, graphics, etc.)
- Languages used to program computers have also changed over time
  - Older languages: Fortran, C, C++
  - Newer "compiled" languages: C#, Java, R
  - Newer "interpreted" languages: Python, JavaScript
- This class is about "principles" of computer programming
  - Common principles behind all languages won't change, even though hardware and languages do

  - How to organize and structure data
  - How to express logical conditions and relations
  - How to solve problems with programs

1.2 Programming Language Concepts

- Machine language
  - Computers are made of electronic circuits
  - Basic instructions are encoded by setting wires to "on" or "off"
    - Read data, write data, add, subtract, etc.
  - Binary digits represent on/off state of wires in a circuit
  - Machine language: which sequence of binary digits (circuit state) represents which computer instruction
    - Example instruction: 0010110010101101
  - Most CPUs use one of two languages: x86 or ARM
- Assembly language
  - Easier way for humans to write machine-language instructions
  - Use a sequence of letters/symbols to represent an instruction, instead of 1s and 0s.
    - Example x86 instruction: `movq %rdx, %rbx`
  - **Assembler**: Translates assembly language code to machine instructions
    - One assembly instruction = one machine-language instruction
    - x86 assembly produces x86 machine code
  - Computers can only execute the machine code
- High-level language
  - More human-readable than assembly language
  - Each statement does not need to correspond to a machine instruction
  - Statements represent more "high-level" concepts, such as storing a value in a variable, not "machine-level" concepts like "read these bits from this address"
  - Most languages we program in are high-level (C, C#, Python…)
  - **Compiler**: Translates high-level language to machine code
    - Small programs in high-level language might produce lots of machine code
    - Compiler is specific to both the source language and the target machine code
  - Compile then execute, since computers can only execute machine code
- Compiled vs. Interpreted languages
  - Not all high-level languages use a compiler - some use an interpreter
  - **Interpreter**: Lets a computer "execute" high-level code by translating one statement at a time to machine code
  - Advantage: Less waiting time before you can run the program (no separate "compile" step)
  - Disadvantage: Program runs slower, since you wait for each high-level statement to be translated before the program can continue
- Managed high-level languages (like C#)
  - Combine features of compiled and interpreted languages
  - Compiler translates high-level statements to **intermediate language** instructions, not machine code
    - Intermediate language: Looks like assembly language, but not specific to any CPU
  - **Runtime** executes compiled program by *interpreting* the intermediate language instructions - translates one at a time to machine code

  - Advantages of managed languages:
    * In a "non-managed" language, a compiled program only works on one OS + CPU combination (**platform**) because it is machine code
    * Managed-language programs can be reused on a different platform without recompiling - intermediate language is not machine code and not CPU-specific
    * Still need to write an intermediate language intepreter for each platform (so it produces the right machine code), but in a non-managed language you must write a compiler for each platform
    * Intermediate-language interpreter is much faster than a high-level language interpreter, so programs run faster than an "interpreted language" like Python
  - This still runs slower than a non-managed language (due to the interpreter), so performance-minded programmers use non-managed compiled languages (e.g. for video games)

1.3 Software Concepts

- Flow of execution in a program
  - Program receives input from some source, e.g. keyboard, mouse, data in files
  - Program uses input to make decisions
  - Program produces output for the outside world to see, e.g. by displaying images on screen, writing text to console, or saving data in files
- Program interfaces
  - **GUI** or Graphical User Interface: Input is from clicking mouse in visual elements on screen (buttons, menus, etc.), output is by drawing onto the screen
  - **CLI** or Command Line Interface: Input is from text typed into "command prompt" or "terminal window," output is text printed at same terminal window
  - This class will use CLI because it's simple, portable, easy to work with – no need to learn how to draw images, just read and write text

1.4 Programming Concepts

- Programming workflow (see flowchart)
  - Writing down specifications
  - Creating the source code
  - Running the compiler
  - Reading the compiler's output, warning and error messages
  - Fixing compile errors, if necessary
  - Running and testing the program
  - Debugging the program, if necessary
- Intepreted language workflow (see flowchart)
  - Writing down specifications
  - Creating the source code
  - Running the program in the interpreter
  - Reading the interpreter's output, determining if there is a syntax (language) error or the program finished executing
  - Editing the program to fix syntax errors
  - Testing the program (once it can run with no errors)
  - Debugging the program, if necessary
  - **Advantages**: Fewer steps between writing and executing, can be a faster cycle
  - **Disadvantages**: All errors happen when you run the program, no distinction between syntax errors (compile errors) and logic errors (bugs in running program)

1.4.0.1 Programming workflow

- Integrated Development Environment (IDE)
  - Combines a text editor, compiler, file browser, debugger, and other tools
  - Helps you organize a programming project
  - Helps you write, compile, and test code in one place
  - Visual Studio terms:
    - Solution: An entire software project, including source code, metadata, input data files, etc.
    - "Build solution": Compile all of your code
    - "Start without debugging": Run the compiled code
    - Solution location: The folder (on your computer's file system) that contains the solution, meaning all your code and the information needed to compile and run it

2 C# Fundamentals

2.1 Introduction to the C# Language

- C# is a managed language (as introduced in previous lecture)
  - Write in a high-level language, compile to intermediate language, run intermediate language in interpreter
  - Intermediate language is called CIL (Common Intermediate Language)
  - Interpreter is called .NET Runtime
  - Standard library is called .NET Framework, comes with the compiler and runtime
- It's widespread and popular
  - 7th most used language on StackOverflow, 5th-most if you discount JavaScript and HTML (which are used for websites, not programs)
  - .NET is the 2nd most used library/framework

2.2 The Object-Oriented Paradigm

- C# is called an "object-oriented" language
  - Programming languages have different *paradigms*: philosophies for organizing code, expressing ideas
  - Object-oriented is one such paradigm, C# uses it
  - Meaning of object-oriented: Program mostly consists of *objects*, which are reusable modules of code
  - Each object contains some data (*attributes*) and some functions related to that data (*methods*)
- Object-oriented terms
  - **Class**: A blueprint or template for an object. Code that defines what kind of data the object will contain and what operations (functions) you will be able to do with that data
  - **Object**: A single instance of a class, containing running code with specific values for the data. Each object is a separate "copy" based on the template given by the class.
  - **Method**: A function that modifies an object. This is code that is defined (written) in the class, but when it runs, it only runs on/for a specific object and modifies that object.
  - **Attribute**: A piece of data stored in an object
- Example objects:
  - "Car" object, represents a car

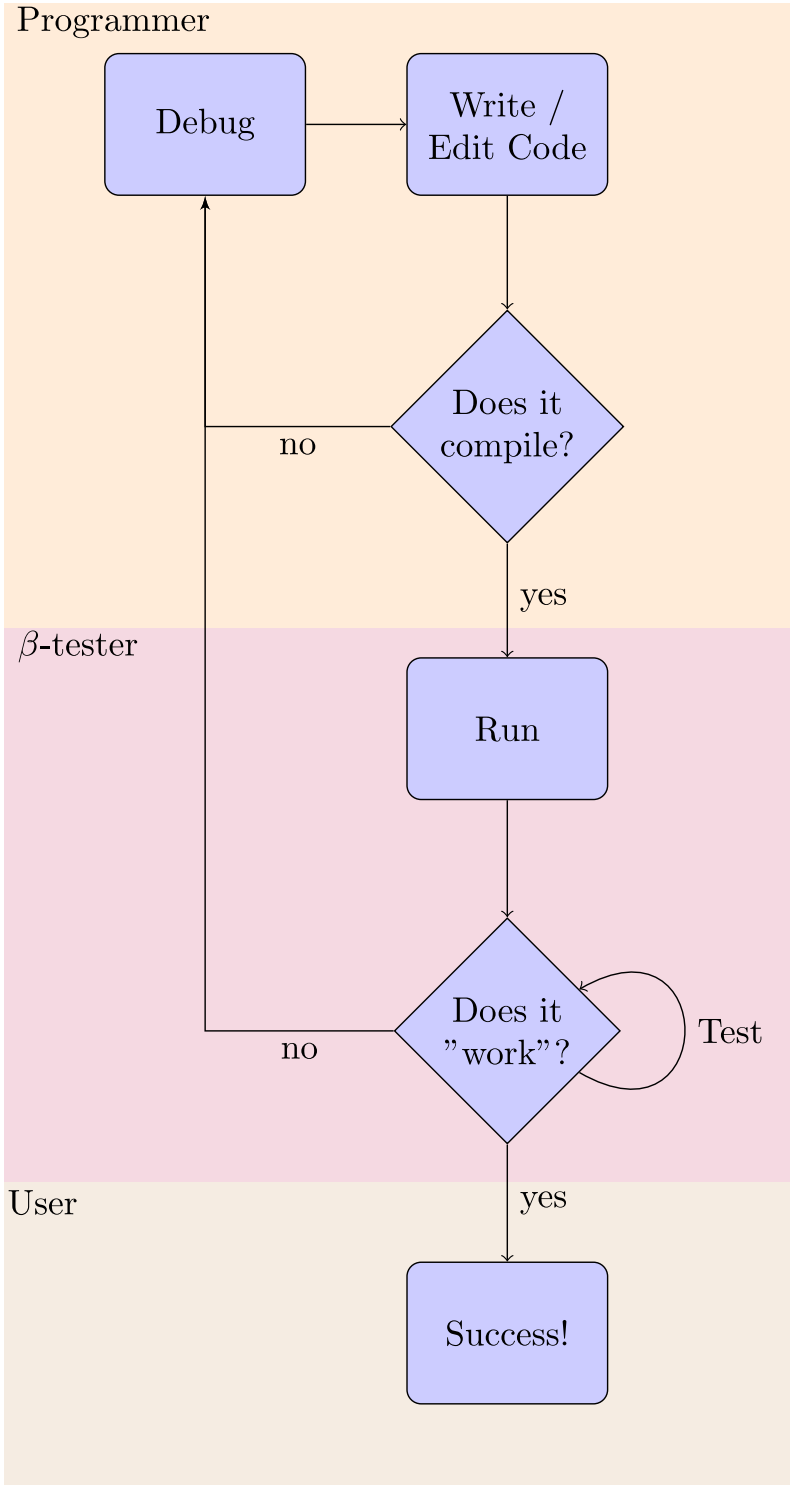

Figure 1: "Flowchart demonstrating roles and tasks of a programmer, beta tester and user in the creation of programs."

        * Attributes: Color, wheel size, engine status (on/off/idle), gear position
        * Methods: Press gas or brake pedal, turn key on/off, shift transmission
    - "Audio" object, represents a song being played in a music player
        * Attributes: Sound wave data, current playback position, target speaker device
        * Methods: Play, pause, stop, fast-forward, rewind

3 First Program

Here's a simple "hello world" program in the C# language:

3.0.0.1 Hello World

```
/* I'm a multi-line comment,
 * I can span over multiple lines!
 */
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Hello, world!"); // I'm an in-line comment.
    }
}
```

Features of this program:

- A multi-line comment: everything between the `/*` and `/*` is considered a *comment*, i.e. text for humans to read. It will be ignored by the C# compiler and has no effect on the program.
- A `using` statement: This imports code definitions from the System *namespace*, which is part of the .NET Framework (the standard library).
    - In C#, code is organized into **namespaces**, which group related classes together
    - If you want to use code from a different namespace, you need a `using` statement to "import" that namespace
    - All the standard library code is in different namespaces from the code you will be writing, so you'll need `using` statements to access it
- A class declaration
    - Syntax: `class [name of class]`, then `{` to begin the body of the class, then `}` to end the body of the class
    - All code between opening `{` and closing `}` is part of the class named by the `class [name]` statement
- A method declaration
    - The name of the method is `Main`, and is followed by empty parentheses (we'll get to those later, but they're required)
    - Just like with the class declaration, after the name, `{` begins the body of the method, `}` ends it
- A statement inside the body of the method
    - This is the part of the program that actually "does something": It prints a line of text to the console
    - A statement *must* end in a semicolon (the class header and method header aren't statements)
    - This statement contains a class name (`Console`), followed by a method name (`WriteLine`). It calls the `WriteLine` method in the `Console` class.

  - The **argument** to the `WriteLine` method is the text "Hello, world!", which is in parentheses after the name of the method. This is the text that gets printed in the console: The `WriteLine` method (which is in the standard library) takes an argument and prints it to the console.
  - Note that the argument to `WriteLine` is inside double-quotes. This means it is a **string**, i.e. textual data, not a piece of C# code. The quotes are required in order to distinguish between text and code.
- An in-line comment: All the text from the `//` to the end of the line is considered a comment, and is ignored by teh C# compiler.

3.1 Rules of C# Syntax

- Each statement must end in a semicolon
  - Class and method declarations are not statements
  - A method *contains* some statements, but it is not a statement
- All words are case-sensitive
  - A class named `Program` is not the same as one named `program`
  - A method named `writeline` is not the same as one named `WriteLine`
- Braces and parentheses must always be matched
  - Once you start a class or method definition with `{`, you must end it with `}`
- Whitespace – spaces, tabs, and newlines – has almost no meaning
  - There must be at least 1 space between words
  - Spaces are counted exactly if they are inside string data, e.g. `"Hello  world!"`
  - Otherwise, entire program could be written on one line; it would have the same meaning
  - Spaces and new lines are just to help humans read the code
- All C# applications must have a `Main` method
  - Name must match exactly, otherwise .NET runtime will get confused
  - This is the first code to run when the application starts – any other code (in methods) will only run when its method is called

3.2 Conventions of C# Programs

- Conventions: Not enforced by the compiler/language, but expected by humans
  - Program will still work if you break them, but other programmers will be confused
- Indentation
  - After a class or method declaration (header), put the opening `{` on a new line underneath it
  - Then indent the next line by 4 spaces, and all other lines "inside" the class or method body
  - De-indent by 4 spaces at end of method body, so ending `}` aligns vertically with opening `{`
  - Method definition inside class definition: Indent body of method by another 4 spaces
  - In general, any code between `{` and `}` should be indented by 4 spaces relative to the `{` and `}`
- Code files
  - C# code is stored in files that end with the extension ".cs"
  - Each ".cs" file contains exactly one class
  - The name of the file is the same as the name of the class (Program.cs contains `class Program`)

3.3 Reserved Words and Identifiers

- Reserved words: Keywords in the C# language
  - Note they have a distinct color in the code sample and in Visual Studio
  - Built-in commands/features of the language
  - Can only be used for one specific purpose; meaning cannot be changed
  - Examples:
    - using
    - class
    - public
    - private
    - namespace
    - this
    - if
    - else
    - for
    - while
    - do
    - return
- Identifiers: Human-chosen names
  - Names for classes, variables, methods, namespaces, etc.
  - Some have already been chosen for the standard library (e.g. Console, WriteLine), but they're still identifiers, not keywords
  - Rules for identifiers:
    - Must not be a reserved word
    - Must contain only letters, numbers, and underscore – no spaces
    - Must not begin with a number
    - Are case sensitive
  - Conventions for identifiers
    - Should be descriptive, e.g. "AudioFile" or "userInput" not "a" or "x"
    - Should be easy for humans to read and type
    - If name is multiple words, use CamelCase to distinguish words
    - Class and method names should start with capitals, e.g. `class AudioFile`
    - Variable names should start with lowercase letters, then capitalize subsequent words, e.g. `myFavoriteNumber`

4 Datatypes and Operators

4.1 Datatypes Nomenclature

- Value types
  - Numeric
    - Signed integer (`sbyte`, `short`, *`int`*, `long`)
    - Unsigned integer (`byte`, `ushort`, `uint`, `ulong`)
    - Real number (`float`, *`double`*, `decimal`)
  - Logical (*`bool`*)
  - Character (*`char`*)
- Reference types

- String (*string*)
- Object (`object`)

(In italics*, the one we will mainly be using.)*

Integers are "whole" numbers (= {..., -1, 0, 1, 2, 3, ...}), floating point numbers are real numbers ( ), strings are "text messages", ...

Please refer to the "Datatypes in C#" cheatsheet[1] for more information about datatypes.

4.2 String and Int Variables

Literals are fixed values ("Hi Mom", 40, 1.2404, ...) in the source code.

4.2.0.1 My First Variables

```
class MyFirstVariables
{
    static void Main()
    {
        // Declaration
        int myAge;
        string myName;

        // Assignment
        myAge = 40;
        myName = "Clément";

        // Displaying
        Console.WriteLine($"I am {myAge} old and my name is {myName}.");
    }
}
```

A variable has a *name* (which must be an identifier), a *type*, a *size*, and a *value*.

Variable Name	Variable Type	Variable Size	Variable Value
`myAge`	`int`	32 bit	`40`
`myName`	`string`	Variable	`"Clément"`

4.3 Variable Initialization

You can declare and assign a variable in one statement using what is called an "initialization statement".

4.3.0.1 initialization

```
string myMessage = "Hey Mom";
int myValue = 12;
```

There is now one additional rule when it comes to chosing a valid identifier for your variable name: you can not take an identifier that was already used. That is, you can have only one variable named `myMessage`: if

[1] https://csci-1301.github.io/datatypes_in_charp.html

you want to re-assign a variable, you can not use an initialization again (that would re-declare the variable), you need to use an assignment statement again.

4.4 Remarks

- The value can change (hence the name!) if you re-assign it. The previously stored value is simply wiped out, and lost.
- You can store one variable's value into another, but that value in the other variable won't change when the original variable's value changes:

```
int a = 12;
int b = a; // b's value is 12
a = 0; // a's value changed to 0, but b's value is still 12.
```

- We can perform basic math operations with numeric datatypes: + (sum), * (multiplication), - (substraction) but also the modulo operation (%), which corresponds to the remainder. More details will be given in lab #3, in homework #2, and during lecture #4.
- There is a difference between

```
int sum = num * 2; // The value of sum is num's value times two
Console.WriteLine($"{sum}"); // The value of sum is displayed.
```

and

```
Console.WriteLine($"{num * 2}"); // The value of num times two is displayed, but the value of num
```

- You can combine multiple declarations, initializations, and even mix both in one statement:

```
int a=0, b, c;  // a, b, c are declared as three int variables, and a's value is set to 0.
```

5 Operators