Solving programming problems

by Hans Wichman

Introduction

- These slides contain an overview with examples of different approaches to solve programming problems, some very practical, some bordering on the philosophical (feel free to skip *those*)
- Note that not *all* of these approaches apply to *every* problem
- Note that some of the items refer to specific lectures and will be explained in more detail during that specific lecture

(Algorithmic) problem solving approaches

- Divide & Conquer
- Acting out the algorithm
- Iterations and minimum viable products
- Experiment freely
- Walk it off
- Spam that console
- Binary exclusion
- Hypothesize
- Watch that rerun
- Using the debugger
- Inspect the callstack
- Visualize your program state
- Trace tables
- Unit Testing
- Pseudocode & Drawings

Divide & Conquer

Divide and conquer

- Problems are either trivial or non trivial:
  - Trivial: they can be converted to code with little trouble
  - Non-trivial: they need to be split into sub problems in order to become trivial
- Keep asking yourself:
  - Is this problem trivial or non trivial?
  - How can I split this problem into smaller problems?
- As an example see the dungeon breakdown in the slides for lecture 1
- The next principles also provide handholds/questions you can ask yourself, while you are trying to make problems 'smaller'

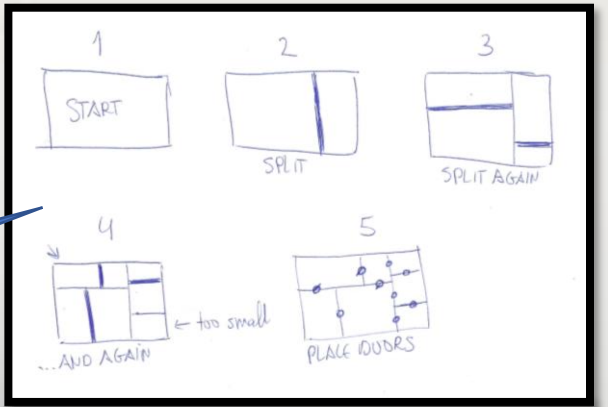
Acting out the algorithm

Acting out the algorithm

If you cannot write down/visualize the algorithm step by step on paper, chances are you cannot implement it

- Step 1 – Step away from the computer
- Step 2 – Act out the algorithm on paper, re-tracing the steps
- Step 3 – Prototype on the computer
- Step 4 – Start over at step 1 if necessary

For example, for the dungeon assignment

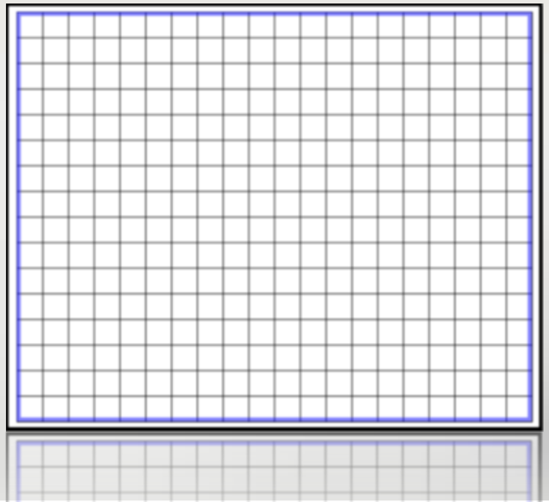


Iteration / MVP

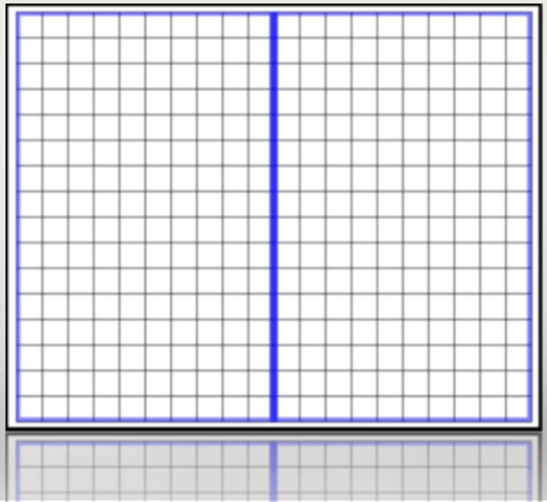
Iteration

- Algorithm execution involves a lot of iteration (eg for loops)
- But algorithm *development* ALSO involves a lot of iteration
- Instead of Divide & Conquer, we could say Iterate & Conquer ☺
- For example, referring to the dungeon problem from assignment 1 again:

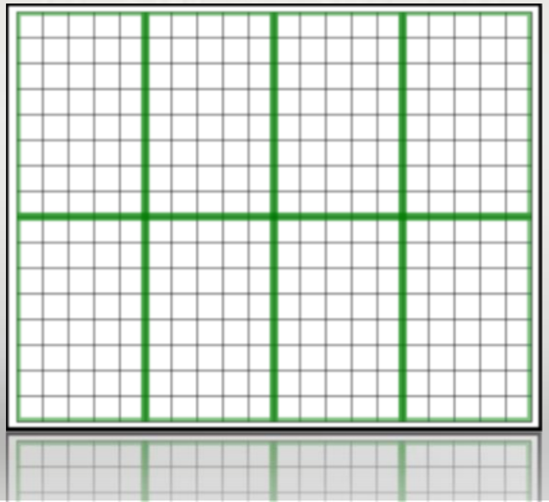
First, simple iteration:
Create 1 room



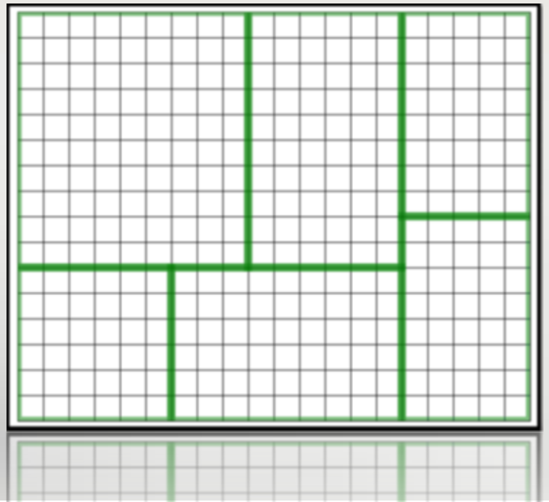
Second, little bit harder:
Split 1 room



Third, a little bit harder:
Split *all* rooms



Endboss:
Split *all* rooms randomly



Minimum viable product

- Iteration is *closely* tied to the concept of Minimum Viable Product
- What is the minimal step I can take, that takes me in the direction of *a* solution?
- What is the *next* minimal step I can take, that takes me a *bit* further?
- Answering questions likes these often drive the iterations of your code (see previous slide)
- At first this may feel like going ultra slow, but speed & stepsize will increase over time

Experiment freely

Experimentation

- Given the input and output requirements of a problem,
  it is important to play around
- For example, given the dungeon size and minimum room size, can you:
  - generate 1 room with the size of the dungeon?
  - explain how the room coordinates work?
  - generate random rooms?
  - subdivide a single room into two rooms?
  - print whether a room meets the minimum size requirement?
  - print whether any rooms have an overlap?
- Constantly think: what else can I try that I haven't tried yet no matter how small
- The mindset of experimentation helps avoiding analysis paralysis

Embrace failure

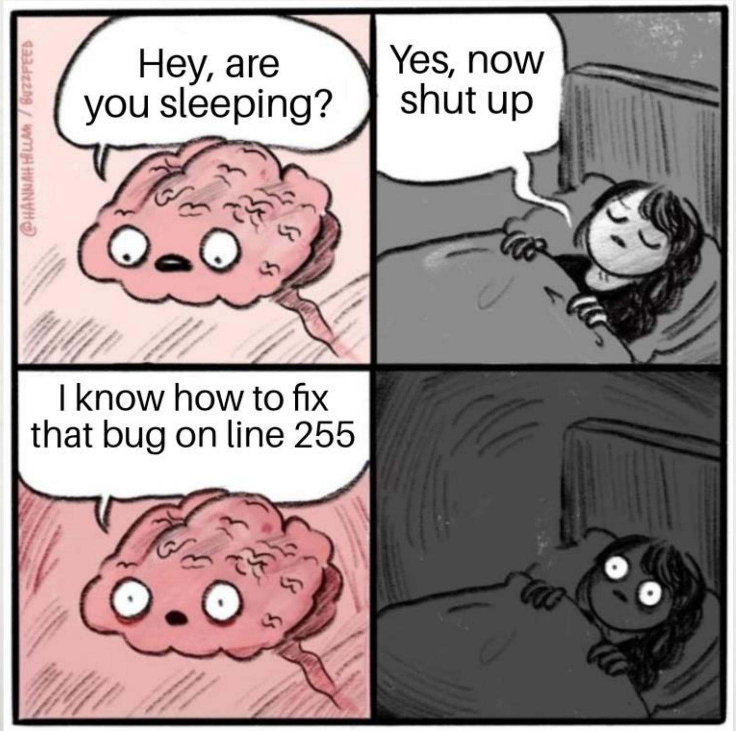
Embrace failure

- This ties in with experimenting freely:
  - Trying to do things right the first time will not help you
  - Trying & failing is part of the game
  - By trying, you find out what works and what doesn't, even if it is just for example randomly changing numbers to see the effect
  - Silence your inner voice
    (unless it's making compliments, but it hardly ever does)

Walk it off

Walk it off

- Problems are sometimes best solved walking or sleeping instead of coding:
  - Take a walk
  - Work on something else
  - Get some sleep
  - Discuss the problem with a friend
  - Explain the problem to your mom/dog/goldfish or buy a rubber duck, name it, and [explain it to the duck]
  - Skip that specific part and work on the next assignment



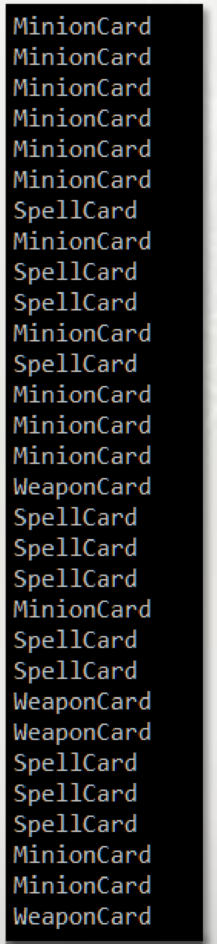
Spam that Console

Console.WriteLine is your 'friend'

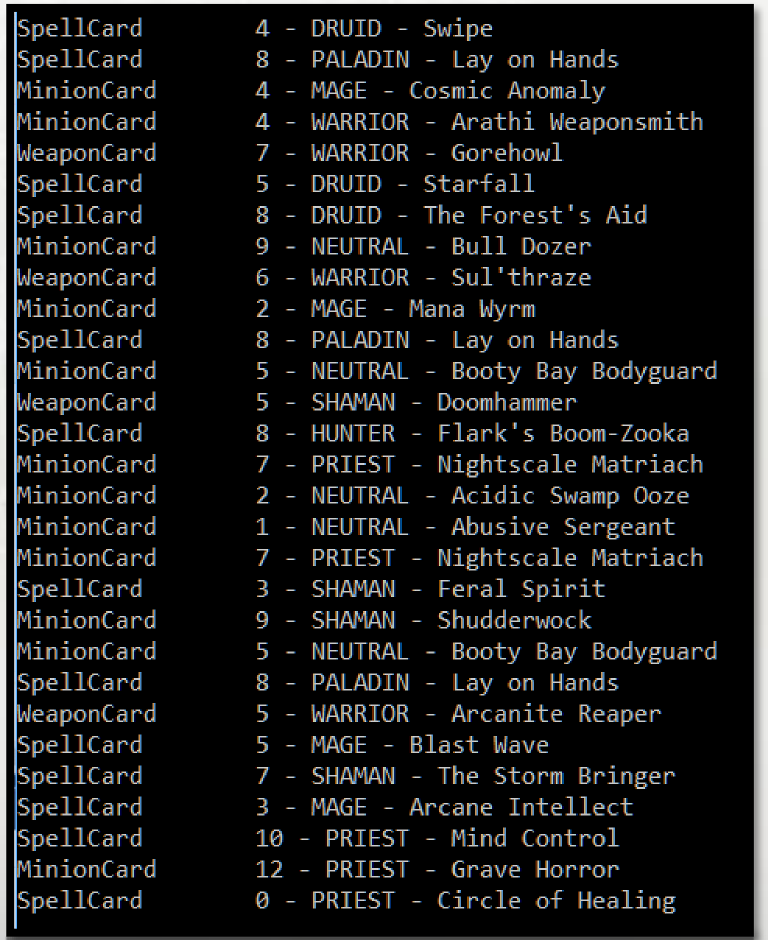
- When stuck you need to gain control/insight/a grip on your code
- One way to do that is through a lot of 'good' debug statements
- What does 'good' mean?
  - Objects should have descriptive correctly implemented ToString methods
  - The debug output should describe the program flow, showing for example:
    - Input data → Algorithm progress → Output data

- Keep improving and changing the debug output, using whitespace, newlines, tabs and indents until you can easily follow the program flow

Example of a bad vs a good ToString method



```
MinionCard
MinionCard
MinionCard
MinionCard
MinionCard
MinionCard
SpellCard
MinionCard
SpellCard
SpellCard
MinionCard
SpellCard
MinionCard
MinionCard
MinionCard
WeaponCard
SpellCard
SpellCard
SpellCard
MinionCard
SpellCard
SpellCard
WeaponCard
WeaponCard
SpellCard
SpellCard
SpellCard
MinionCard
MinionCard
WeaponCard
```



```
SpellCard        4 - DRUID - Swipe
SpellCard        8 - PALADIN - Lay on Hands
MinionCard       4 - MAGE - Cosmic Anomaly
MinionCard       4 - WARRIOR - Arathi Weaponsmith
WeaponCard       7 - WARRIOR - Gorehowl
SpellCard        5 - DRUID - Starfall
SpellCard        8 - DRUID - The Forest's Aid
MinionCard       9 - NEUTRAL - Bull Dozer
WeaponCard       6 - WARRIOR - Sul'thraze
MinionCard       2 - MAGE - Mana Wyrm
SpellCard        8 - PALADIN - Lay on Hands
MinionCard       5 - NEUTRAL - Booty Bay Bodyguard
WeaponCard       5 - SHAMAN - Doomhammer
SpellCard        8 - HUNTER - Flark's Boom-Zooka
MinionCard       7 - PRIEST - Nightscale Matriach
MinionCard       2 - NEUTRAL - Acidic Swamp Ooze
MinionCard       1 - NEUTRAL - Abusive Sergeant
MinionCard       7 - PRIEST - Nightscale Matriach
SpellCard        3 - SHAMAN - Feral Spirit
MinionCard       9 - SHAMAN - Shudderwock
MinionCard       5 - NEUTRAL - Booty Bay Bodyguard
SpellCard        8 - PALADIN - Lay on Hands
WeaponCard       5 - WARRIOR - Arcanite Reaper
SpellCard        5 - MAGE - Blast Wave
SpellCard        7 - SHAMAN - The Storm Bringer
SpellCard        3 - MAGE - Arcane Intellect
SpellCard        10 - PRIEST - Mind Control
MinionCard       12 - PRIEST - Grave Horror
SpellCard        0 - PRIEST - Circle of Healing
```

Advanced logging options

- Conditionals (see hearthstone example in lecture 2)
- Using logging frameworks

Binary exclusion

Binary exclusion

- The binary exclusion/elimination principle looks a lot like the 'guess-a-number' game, where you can only answer higher or lower
- When programming, the principle is this:
  - If your code breaks, or you are stuck on a problem, comment out roughly 50% of your code
  - If the problem disappears, it was in the 'disabled' code
  - If the problem is still there, it is in the code that is still 'enabled'
  - Keep repeating this process until you've narrowed down the problem (And then start debugging that piece of code → see next slides)
- Of course in a OO program this is not always that easy to do

Hypothesize

Hypothesize

- Brainstorm about what the problem could be
- Sort the answers based on likelihood
- Think about tests you can execute to validate each hypothesis

- Word of warning:
  - In my own experience, on multiple occasions in the past,
    the least likely 'cause' turned out to be the culprit ☺. In other words, alternatively, you could also ask yourself: what will definitely ***not*** be the problem and try that first.

Watching reruns

Watching reruns

- Your program breaks so you run it again to see if the problem magically disappeared. It didn't. You'll try again and the problem will still be there. It's okay, we all do that ☺
- Although running and re-running doesn't fix your problem, it does allow you to try and test as many different things as possible. Change a number here and there, a +1 into a -1, a > into a >= or < and see what happens.
- Just make sure that after fixing the problem, you understand the why and how. That is the difference between Programming by Coincidence, and actually knowing what you are doing.

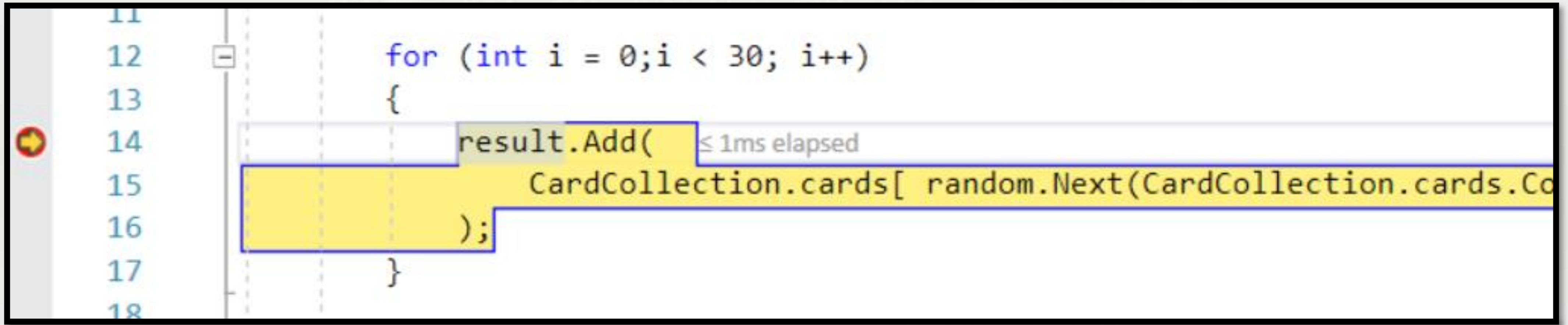
Watching reruns

And yes, there are memes about this, in case you were wondering ☺

Using the debugger

Using the debugger

- Your IDE's debugger allows you to set *multiple* so called *breakpoints*, usually by clicking on or next to the line numbers (differs per IDE):

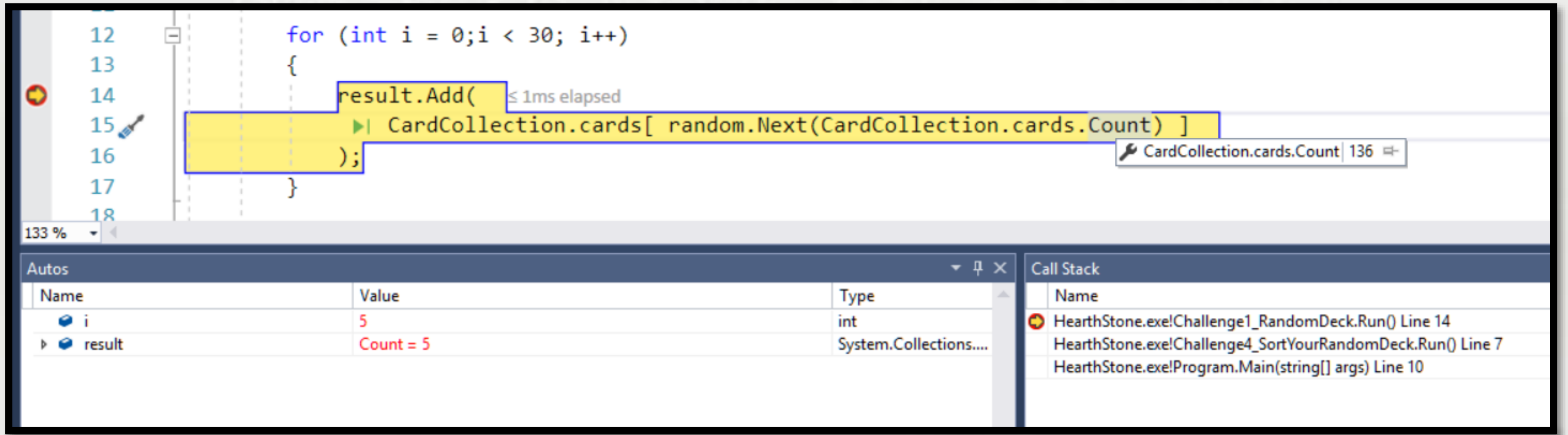


```
for (int i = 0;i < 30; i++)
{
    result.Add(    ≤ 1ms elapsed
        CardCollection.cards[ random.Next(CardCollection.cards.Co
    );
}
```

- A breakpoint is indicated with a red dot

Using the debugger

- While running your code in ***debug*** mode, the program will halt at that breakpoint, allowing you to inspect all variables:



```
for (int i = 0;i < 30; i++)
{
    result.Add(    ≤ 1ms elapsed
        CardCollection.cards[ random.Next(CardCollection.cards.Count) ]
    );
}
```

CardCollection.cards.Count | 136

133 %

Autos

Name	Value	Type
i	5	int
result	Count = 5	System.Collections....

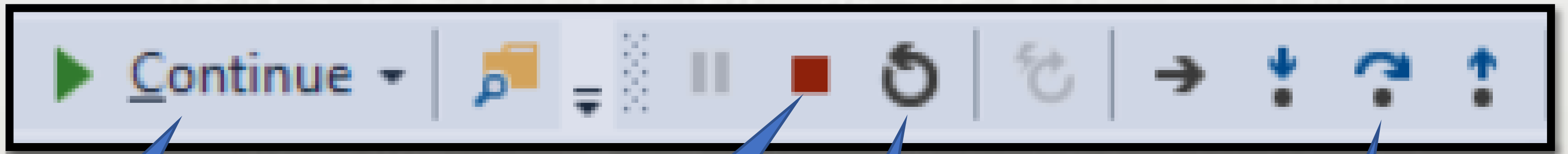
Call Stack

Name
HearthStone.exe!Challenge1_RandomDeck.Run() Line 14
HearthStone.exe!Challenge4_SortYourRandomDeck.Run() Line 7
HearthStone.exe!Program.Main(string[] args) Line 10

- Note the value of *i* and *result*, the value of *cards.Count* under the mouse and the call stack on the right. You can click/expand all of these.

Using the debugging toolbar 1/2

- While in debug mode, you can use the debug toolbar to control the debug process:



Start running the code again until the next breakpoint

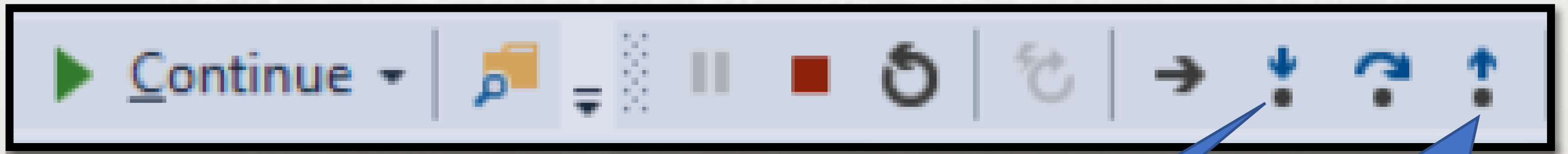
Stop the program

Restart the program

Move to the next line

Using the debugging toolbar 2/2

- For deeper inspection you can also 'follow' the program flow through methods call using 'Step into' and 'Step out':



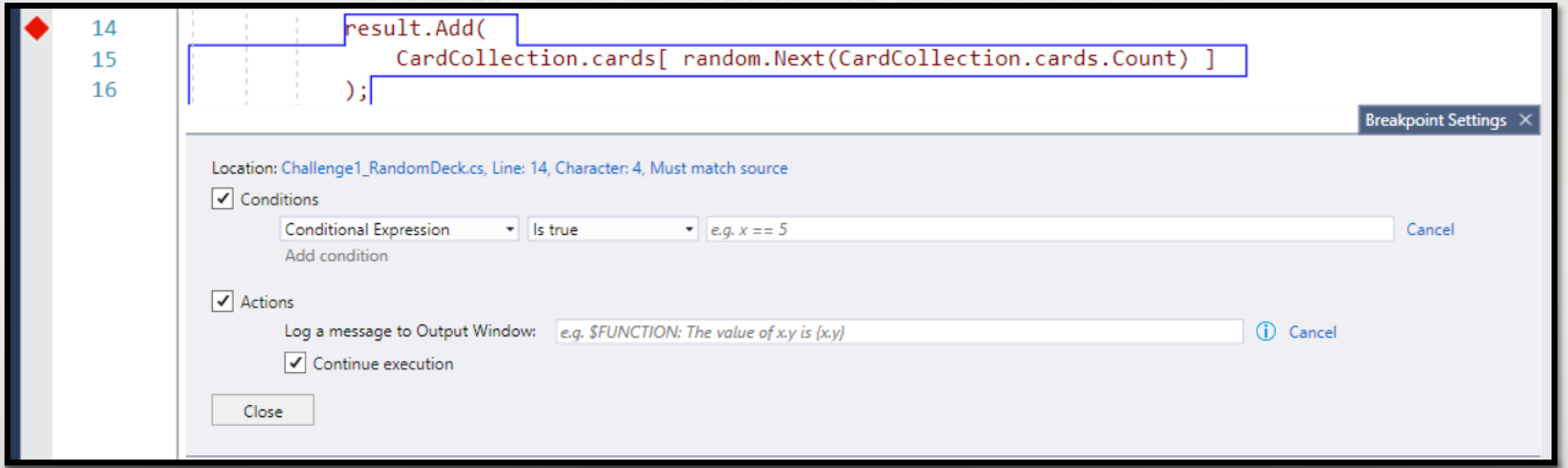
Instead of just executing the next line as a single monolithic whole, the debugger will jump into the current method call (if applicable) to continue debugging there.

To exit the current level of detail, you can press this button and continue at the parent. Try these options with your own nested methods for better understanding.

Using the debugger

MORE INFO

- For more control you can:
  - enable/disable breakpoints while your code is running
  - specify specific conditions and actions for your breakpoint (and combine this with logging!)



```
result.Add(
    CardCollection.cards[ random.Next(CardCollection.cards.Count) ]
);
```

Breakpoint Settings

Location: Challenge1_RandomDeck.cs, Line: 14, Character: 4, Must match source

✓ Conditions

Conditional Expression | Is true | e.g. x == 5 | Cancel

Add condition

✓ Actions

Log a message to Output Window: e.g. $FUNCTION: The value of x.y is {x.y} | Cancel

✓ Continue execution

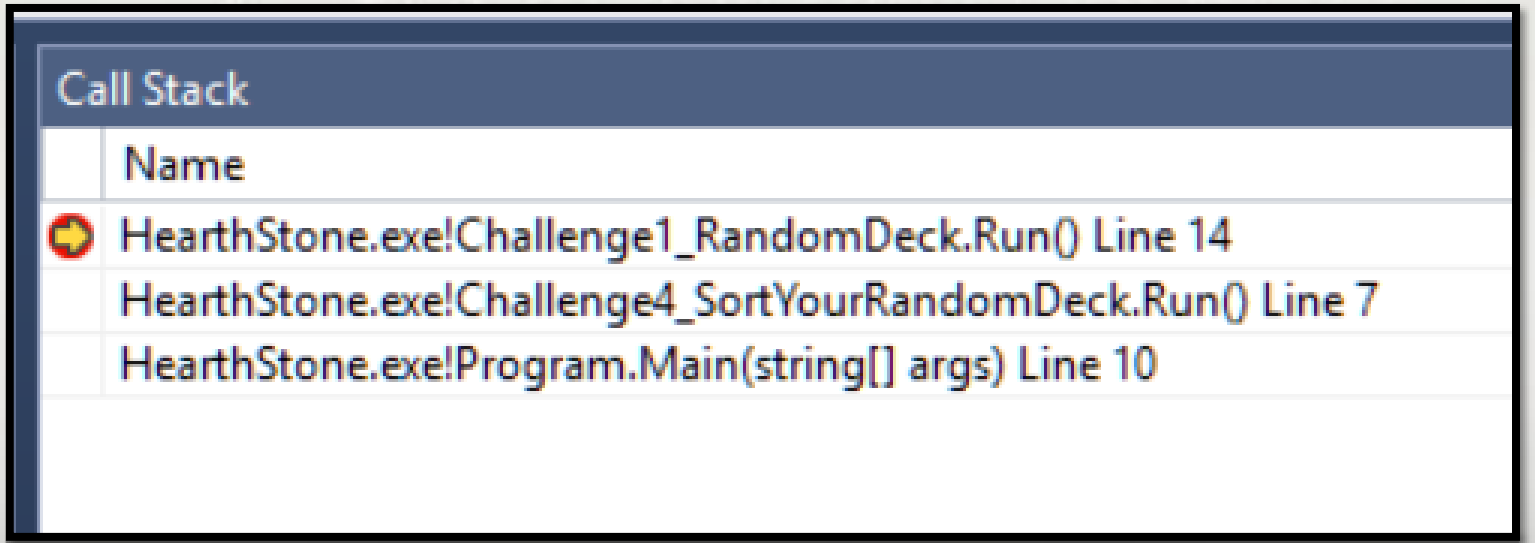
Close

Inspect the Call Stack

Inspect the Call Stack

MORE INFO

- Every method call is put onto the Call Stack (if you don't know what this means → check out the C# Essential Slides on BlackBoard)
- The moment your program crashes or hits a breakpoint, you can inspect this call stack and double click calls to inspect the program flow:

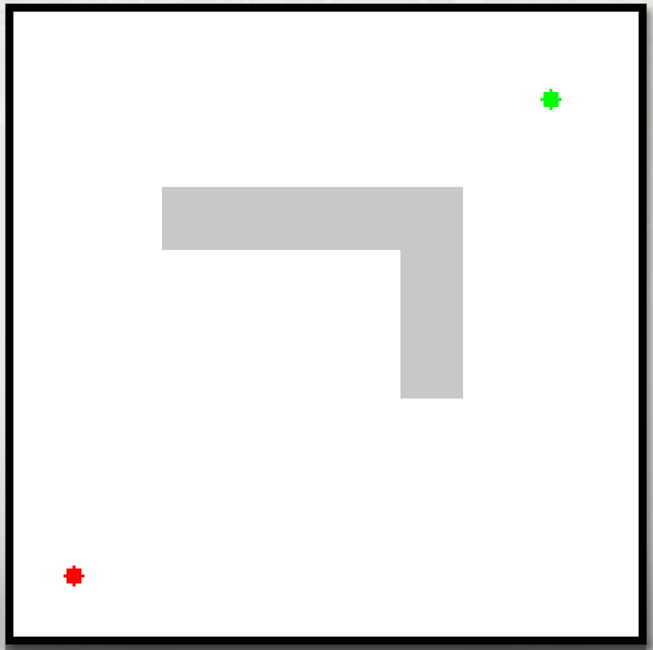


```
HearthStone.exe!Challenge1_RandomDeck.Run() Line 14
HearthStone.exe!Challenge4_SortYourRandomDeck.Run() Line 7
HearthStone.exe!Program.Main(string[] args) Line 10
```

Visualize your program state

Visualize your program state

- Debugging is great, printing debug info as well, but sometimes you need to visualize things (maybe even step by step) to actually get a grip on what is happening, for example during a step by step pathfinding algorithm (more about this in lecture 4):



Visualize your program state

- There are two basic ways to do that for the assignments:
  1. Do not execute the algorithm in one go, but trigger a next step on keypress
     For example instead of calling 'Generate' you would implement:
     Reset() & Step(), where Reset() initializes the algorithm and Step() executes one step of your algorithm when you press a key and redraws.
     An example of this will be shown during one of the lectures.
  2. A dirty trick to do the same will less hassle is using Threads, as in:
     new Thread (myAlgorithm.Generate).Start();
     and use *Thread.Sleep(1000); redraw();* in your algorithm loop.
     Note that the program might crash now and then, and you should never do this in production code, but for quick and dirty debugging it's fine.
     More info in threading will be provided during the networking course.

Trace tables

Trace tables

- Trace tables keep track of all important algorithmic variables through the iterations of the algorithm line by line
- Depending on the specific algorithm this might be very easy to do, or next to impossible.
- In general it is very worthwhile to:
  - try to predict the output of the algorithm based on well chosen input (well chosen is 'leading to output that you can verify to be correct by hand')
  - execute the algorithm
  - compare the algorithm values/output with your predicted values

Trace table example

MORE INFO

```
int result = 0;
for (int i = 0; i < 4; i++) {
        if (result % 2 == 0) result = result + i*4 + 1;
        else result = result * 2;
}
```

Start*	**-**	**result = 0**
Iteration 1	i = 0	result = 1
Iteration 2	i = 1	result = 2
Iteration 3	i = 2	result = 11
Iteration 4	i = 3	result = 22

* Note that we could also do before and after values or a line by line approach as can be found under the 'more info' button

Unit testing

Unit testing

MORE INFO

This is a large topic, which we'll get into more in the second year.
You had a taste of it during Physics Programming and the idea with unit testing is that you test the smallest units of your code to verify they are correct.

What you *haven't* seen yet is that there are actually unit testing frameworks to help you do that and make this easier on you, but the principle remains the same:

Verify that the foundations on which you are building your code are indeed functioning correctly. Through assert statements, through unit testing frameworks, through Console.WriteLine (myValue == expectedValue) etc.

Pseudocode and drawings

Pseudocode and drawings

- Pseudo code and drawings (sketches, flowcharts, doodles, etc) can help you describe and gain more clarity on your algorithm. For more information please refer to the "Describing algorithms" slides on blackboard.