Bug Catching: Automated Program Verification
15414/15614 Spring 2021
Lecture 1: Introduction

Matt Fredrikson and Frank Pfenning

January 18, 2022

Ohyay

- Refresh the page if a technical issue arises
- Communicate via chat, questions, or with emojis
- Or via raise your hand boxes
  - Click on a box
  - Unmute yourself
  - Speak ...
  - Click box again to disappear
- Breakout rooms
- Office hours
- Help Desk!

Course staff

Instructors

Frank Pfenning
($\sim$ Parts I & II)

Matt Fredrikson
($\sim$ Part III)

Teaching Assistants

May Li
Warwick Marangos
Long Pham
Victor Song

Learning objectives

For this lecture

- What is this course about?
- What are the learning objectives for the course?
- How does it fit into the curriculum?
- How does the course work?
- Remember . . .

Bad code

- **April, 2014** OpenSSL announced critical vulnerability in their implementation of the Heartbeat Extension.

Bad code

- **April, 2014** OpenSSL announced critical vulnerability in their implementation of the Heartbeat Extension.
- "The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software."

Bad code

- **April, 2014** OpenSSL announced critical vulnerability in their implementation of the Heartbeat Extension.
- "The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software."
- "...this allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users."

Heartbleed, explained

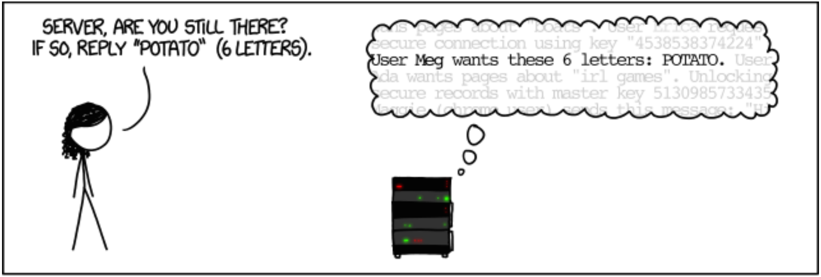


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

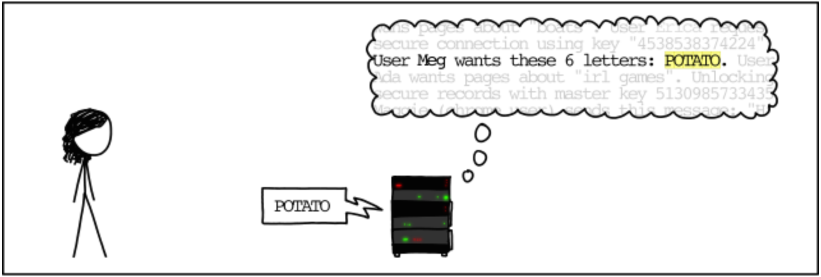


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

Image source: Randall Munroe, xkcd.com

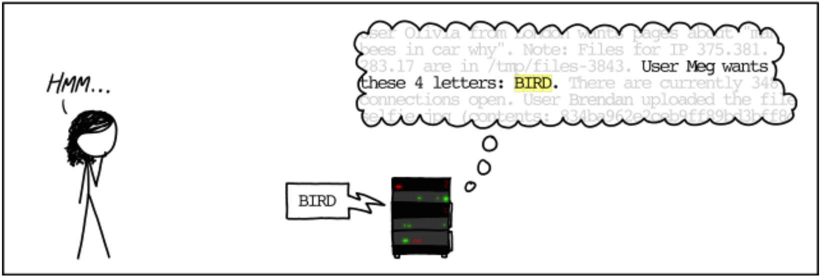


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

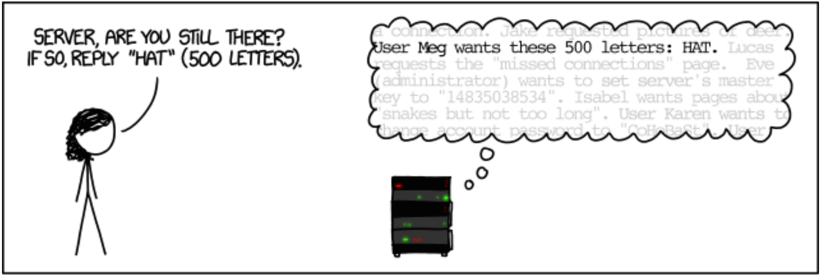


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

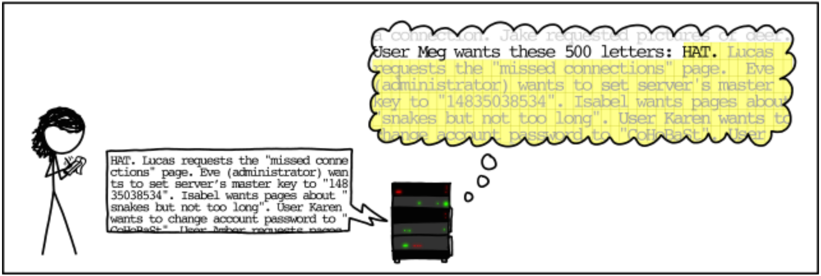


Image source: Randall Munroe, xkcd.com

Algorithms vs. code

```
int binarySearch(int key, int[] a, int n) {
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = (low + high) / 2;

        if(a[mid] == key) return mid; // key found
        else if(a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return -1;  // key not found.
}
```

Code matters

This is a correct binary search algorithm

Code matters

This is a correct binary search algorithm

But what if `low` $+$ `high` $> 2^{31} - 1$?

Code matters

This is a correct binary search algorithm

But what if `low` $+$ `high` $> 2^{31} - 1$?

Then `mid` $=$ (`low` $+$ `high`) / 2 becomes negative

Code matters

This is a correct binary search algorithm

But what if `low` $+$ `high` $> 2^{31} - 1$?

Then `mid` $=$ (`low` $+$ `high`) / 2 becomes negative

- Best case: `ArrayIndexOutOfBoundsException`

Code matters

This is a correct binary search algorithm

But what if `low` $+$ `high` $>$ $2^{31} - 1$?

Then `mid` $=$ (`low` $+$ `high`) / 2 becomes negative

- Best case: `ArrayIndexOutOfBoundsException`
- Worst case: undefined (that is, arbitrary) behavior

Code matters

This is a correct binary search algorithm

But what if `low` $+$ `high` $> 2^{31} - 1$?

Then `mid` $=$ `(low` $+$ `high)` / 2 becomes negative

- Best case: `ArrayIndexOutOfBoundsException`
- Worst case: undefined (that is, arbitrary) behavior

Algorithm may be correct—but we run code, not algorithms.

How do we fix it?

The culprit: `mid = (low + high) / 2`

How do we fix it?

The culprit: `mid = (low + high) / 2`

Need to make sure we don't overflow at any point

How do we fix it?

The culprit: `mid = (low + high) / 2`

Need to make sure we don't overflow at any point

Solution: `mid = low + (high - low)/2`

The fix

```
int binarySearch(int key, int[] a, int n) {
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = low + (high - low) / 2;

        if(a[mid] == key) return mid; // key found
        else if(a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return -1;  // key not found.
}
```

The fix

```
int binarySearch(int key, int[] a, int n)
//@requires 0 <= n && n <= \length(A);
{
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = low + (high - low) / 2;

        if(a[mid] == key) return mid; // key found
        else if(a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return -1;  // key not found.
}
```

The fix

```
int binarySearch(int key, int[] a, int n)
//@requires 0 <= n && n <= \length(a);
/*@ensures (\result == -1 && !is_in(key, A, 0, n))
  @      || (0 <= \result && \result < n
  @          && A[\result] == key); @*/
{
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = low + (high - low) / 2;

        if(a[mid] == key) return mid; // key found
        else if(a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return -1;  // key not found.
}
```

The fix

```
int binarySearch(int key, int[] a, int n)
//@requires 0 <= n && n <= \length(a);
//@requires is_sorted(a, 0, n);
/*@ensures (\result == -1 && !is_in(key, A, 0, n))
  @      || (0 <= \result && \result < n
  @          && A[\result] == key); @*/
{
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = low + (high - low) / 2;

        if(a[mid] == key) return mid; // key found
        else if(a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return -1;  // key not found.
}
```

How do we know if it's correct?

How do we know if it's correct?

One solution: testing

How do we know if it's correct?

One solution: testing

- Probably incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing not feasible

How do we know if it's correct?

One solution: testing

- Probably incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing not feasible

Another: code review

How do we know if it's correct?

One solution: testing

- Probably incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing not feasible

Another: code review

- Correctness definitely important, but not the only thing
- Humans are fallible, bugs are subtle
- What's the specification?

How do we know if it's correct?

One solution: testing

- Probably incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing not feasible

Another: code review

- Correctness definitely important, but not the only thing
- Humans are fallible, bugs are subtle
- What's the specification?

Better: prove correctness

$$\textit{Specification} \iff \textit{Implementation}$$

How do we know if it's correct?

One solution: testing

- Probably incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing not feasible

Another: code review

- Correctness definitely important, but not the only thing
- Humans are fallible, bugs are subtle
- What's the specification?

Better: prove correctness

$$\mathit{Specification} \Longleftrightarrow \mathit{Implementation}$$

- Specification must be precise (many subtleties)
- Meaning of code must be well-defined (many subtleties)
- Reasoning must be sound (many subtleties)

Functional programming

Functional programming

Correctness and proof is not limited to imperative programs!

```
(* balance : rbt -> rbt
 * balance T ==> R
 * REQUIRES T ordered, bh(T) def,
 *   either T is r/b
 *   or T is black; and at most one child is almost r/b
 * ENSURES R is r/b, bh(T) = bh(R), R ordered,
 *   set(T) = set(R)
 *)
fun balance (Blk(Red(Red(a,x,b),y,c),z,d)) =
            Red(Blk(a,x,b),y,Blk(c,z,d))
  | balance (Blk(Red(a,x,Red(b,y,c)),z,d)) =
            Red(Blk(a,x,b),y,Blk(c,z,d))
  | balance (Blk(a,x,Red(b,y,Red(c,z,d)))) =
            Red(Blk(a,x,b),y,Blk(c,z,d))
  | balance (Blk(a,x,Red(Red(b,y,c),z,d))) =
            Red(Blk(a,x,b),y,Blk(c,z,d))
  | balance p = p
```

Different traditions and techniques

Different traditions and techniques

Functional programming: dependent types

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- Properties are expressed in contracts

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- Properties are expressed in contracts
- Reduce correctness to logical propositions (verification condition)

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- Properties are expressed in contracts
- Reduce correctness to logical propositions (verification condition)
- Use automated theorem provers to prove VC

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- Properties are expressed in contracts
- Reduce correctness to logical propositions (verification condition)
- Use automated theorem provers to prove VC

Why3 (this course) supports both!

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- Properties are expressed in contracts
- Reduce correctness to logical propositions (verification condition)
- Use automated theorem provers to prove VC

Why3 (this course) supports both!

- Functional and imperative code in WhyML

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- Properties are expressed in contracts
- Reduce correctness to logical propositions (verification condition)
- Use automated theorem provers to prove VC

Why3 (this course) supports both!

- Functional and imperative code in WhyML
- Automated provers for VC (Z3, CVC, alt-ergo, ...)

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- Properties are expressed in contracts
- Reduce correctness to logical propositions (verification condition)
- Use automated theorem provers to prove VC

Why3 (this course) supports both!

- Functional and imperative code in WhyML
- Automated provers for VC (Z3, CVC, alt-ergo, ...)
- Interactive provers for VC (Coq)

We focus on automated proving

Algorithmic approaches

Formal proofs are tedious

Automatic methods can:

Algorithmic approaches

Formal proofs are tedious

Automatic methods can:

- Check our work
- Fill in low-level details
- Give diagnostic info

Algorithmic approaches

Formal proofs are tedious

Automatic methods can:

- Check our work
- Fill in low-level details
- Give diagnostic info

They usually cannot:

Algorithmic approaches

Formal proofs are tedious

Automatic methods can:

- Check our work
- Fill in low-level details
- Give diagnostic info

They usually cannot:

- Verify "everything" for us
- Generate specification, invariants
- Tell us how to fix bugs

Algorithmic approaches

Formal proofs are tedious

Automatic methods can:

- Check our work
- Fill in low-level details
- Give diagnostic info

They usually cannot:

- Verify “everything” for us
- Generate specification, invariants
- Tell us how to fix bugs

This is what you will learn!

- Make use of these methods
- How (and when) they work

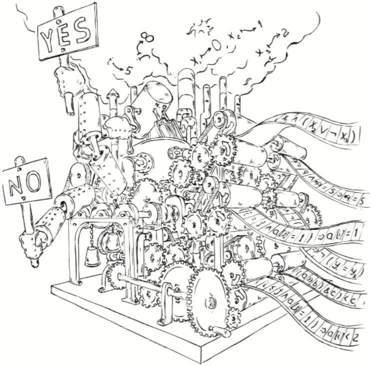


Image source: Daniel Kroening & Ofer Strichman, *Decision Procedures*

Course objectives

Course objectives

- Identify and formalize program correctness

Course objectives

- Identify and formalize program correctness
- Understand language semantics

Course objectives

- Identify and formalize program correctness
- Understand language semantics
- Apply mathematical reasoning to program correctness

Course objectives

- Identify and formalize program correctness
- Understand language semantics
- Apply mathematical reasoning to program correctness
- Learn how to write correct software, from beginning to end

Course objectives

- Identify and formalize program correctness
- Understand language semantics
- Apply mathematical reasoning to program correctness
- Learn how to write correct software, from beginning to end
- Use automated tools that assist verifying your code

Course objectives

- Identify and formalize program correctness
- Understand language semantics
- Apply mathematical reasoning to program correctness
- Learn how to write correct software, from beginning to end
- Use automated tools that assist verifying your code
- Understand how verification tools work

Course objectives

- Identify and formalize program correctness
- Understand language semantics
- Apply mathematical reasoning to program correctness
- Learn how to write correct software, from beginning to end
- Use automated tools that assist verifying your code
- Understand how verification tools work
- Make you better programmers

Course outline

Course outline

Part I: Reasoning about programs: from 122 and 150 to 414

- Gain intuitive understanding of language and methodology

Course outline

Part I: Reasoning about programs: from 122 and 150 to 414

- Gain intuitive understanding of language and methodology

Part II: From inform to formal reasoning

- Specifying meaning of programs
- Specifying meaning of propositions
- Formal reasoning and its justification

Course outline

Part I: Reasoning about programs: from 122 and 150 to 414

- Gain intuitive understanding of language and methodology

Part II: From inform to formal reasoning

- Specifying meaning of programs
- Specifying meaning of propositions
- Formal reasoning and its justification

Part III: Mechanized reasoning

- Techniques for automated proving

Reasoning about correctness

Reasoning about correctness

Functional Correctness

- Specification
- Proof

Reasoning about correctness

Functional Correctness

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise

Reasoning about correctness

Functional Correctness

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise

Systematic proof techniques

- Derived from semantics
- Exhaustive proof rules
- Automatable*

Reasoning about correctness

Functional Correctness

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise

Systematic proof techniques

- Derived from semantics
- Exhaustive proof rules
- Automatable*

```
int[] array_copy(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures \length(\result) == n;
{
  int[] B = alloc_array(int, n);

  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i;
  {
    B[i] = A[i];
  }

  return B;
}
```

Reasoning about correctness

Functional Correctness

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise

Systematic proof techniques

- Derived from semantics
- Exhaustive proof rules
- Automatable*

```
int[] array_copy(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures \length(\result) == n;
{
  int[] B = alloc_array(int, n);

  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i;
  {
    B[i] = A[i];
  }

  return B;
}
```

But . . .

Why3

Why3

Deductive verification platform

- Programming language (WhyML, derived from OCaml)
- Verification toolchain

Why3

Deductive verification platform

- Programming language (WhyML, derived from OCaml)
- Verification toolchain

Rich specification language

- Pre- and post-conditions, loop invariants, assertions
- Pure mathematical functions
- Termination metrics

Why3

Deductive verification platform

- Programming language (WhyML, derived from OCaml)
- Verification toolchain

Rich specification language

- Pre- and post-conditions, loop invariants, assertions
- Pure mathematical functions
- Termination metrics

Programmer writes specification, partial annotations

Why3

Deductive verification platform

- Programming language (WhyML, derived from OCaml)
- Verification toolchain

Rich specification language

- Pre- and post-conditions, loop invariants, assertions
- Pure mathematical functions
- Termination metrics

Programmer writes specification, partial annotations

Compiler proves correctness

Why3

Deductive verification platform

- Programming language (WhyML, derived from OCaml)
- Verification toolchain

Rich specification language

- Pre- and post-conditions, loop invariants, assertions
- Pure mathematical functions
- Termination metrics

Programmer writes specification, partial annotations

Compiler proves correctness

When it works! (It's not quite like a type-checker ...)

Automated verifiers

Systems that prove that programs match their specifications

Basic idea:

1. Translate programs into *proof obligations*
2. Encode proof obligations as satisfiability
3. Solve using a decision procedure

Automated verifiers

Systems that prove that programs match their specifications

Problem is undecidable!

1. Prover needs "hints" from programmer
2. Finding the right set of hints can be challenging

Basic idea:

1. Translate programs into *proof obligations*
2. Encode proof obligations as satisfiability
3. Solve using a decision procedure

Automated verifiers

Systems that prove that programs match their specifications

Problem is undecidable!

1. Prover needs "hints" from programmer
2. Finding the right set of hints can be challenging

Verifiers are complex systems

- We'll deep-dive into selected components
- Understand "big picture" for the rest

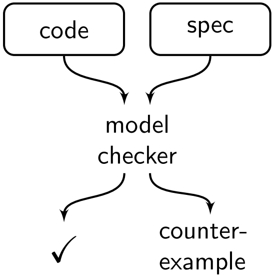
Basic idea:

1. Translate programs into *proof obligations*
2. Encode proof obligations as satisfiability
3. Solve using a decision procedure

Model Checking

Fully-automatic techniques for finding bugs (or proving their absence)

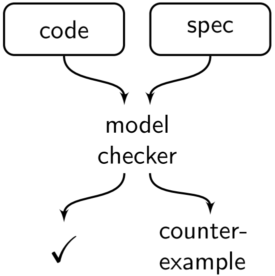
- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples



Model Checking

Fully-automatic techniques for finding bugs (or proving their absence)

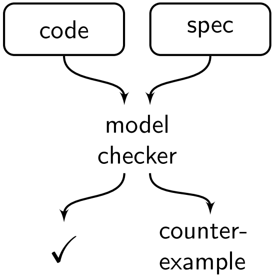
- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples
- No proofs!



Model Checking

Fully-automatic techniques for finding bugs (or proving their absence)

- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples
- No proofs!
- **Downside**: "State explosion"

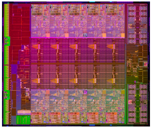


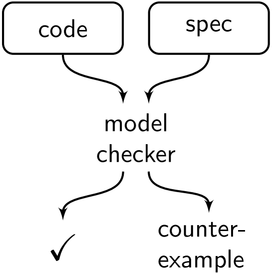
Model Checking

Fully-automatic techniques for finding bugs (or proving their absence)

- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples
- No proofs!
- **Downside**: "State explosion"

10^{70} atoms 10^{500000} states





Model Checking

Clever ways of dealing with state explosion:

Model Checking

Clever ways of dealing with state explosion:

- Partial order reduction
- Bounded model checking
- Symbolic representations
- Abstraction & refinement

Model Checking

Clever ways of dealing with state explosion:

- Partial order reduction
- Bounded model checking
- Symbolic representations
- Abstraction & refinement

Now widely used for bug-finding:

- Hardware, software, protocols, ...
- Microsoft, Intel, Amazon, Google, NASA, ...

Model Checking

Clever ways of dealing with state explosion:

- Partial order reduction
- Bounded model checking
- Symbolic representations
- Abstraction & refinement

Now widely used for bug-finding:

- Hardware, software, protocols, ...
- Microsoft, Intel, Amazon, Google, NASA, ...

Ed Clarke, 1945–2020
Turing Award, 2007

Model Checking

Clever ways of dealing with state explosion:

- Partial order reduction
- Bounded model checking
- Symbolic representations
- Abstraction & refinement

Now widely used for bug-finding:

- Hardware, software, protocols, ...
- Microsoft, Intel, Amazon, Google, NASA, ...

Ed Clarke, 1945–2020
Turing Award, 2007
First developed this course!

Grading

Breakdown:

- 50% assignments
  (written + programming)
- 15% mini-project 1
- 15% mini-project 2
- 20% final exam

6 assignments
done individually

2 mini-projects
pick from small menu
can work with a partner

Participation:

- Come to lecture
- Answer questions
  (in class and on Piazza!)
- Contribute to discussion

Written parts of assignments

Written homeworks focus on theory and fundamental skills

Written parts of assignments

Written homeworks focus on theory and fundamental skills

Grades are based on:

- Correctness of your answer
- How you present your reasoning

Written parts of assignments

Written homeworks focus on theory and fundamental skills

Grades are based on:

- Correctness of your answer
- How you present your reasoning

Strive for **clarity & conciseness**

- Show each step of your reasoning
- State your assumptions
- Answers without these $\longrightarrow$ no points

Programming parts of assignments

For the programming, you will:

- Implement some functionality (data structure or algorithm)
- Specify correctness for that functionality
- Use Why3 to prove it correct

Programming parts of assignments

For the programming, you will:

- Implement some functionality (data structure or algorithm)
- Specify correctness for that functionality
- Use Why3 to prove it correct

Most important criterion is **correctness**.

Programming parts of assignments

For the programming, you will:

- Implement some functionality (data structure or algorithm)
- Specify correctness for that functionality
- Use Why3 to prove it correct

Most important criterion is **correctness**.

Full points when you provide the following

- Correct implementation
- Correct specification
- Correct contracts
- Sufficient contracts for verification

Programming parts of assignments

For the programming, you will:

- Implement some functionality (data structure or algorithm)
- Specify correctness for that functionality
- Use Why3 to prove it correct

Most important criterion is **correctness**.

Full points when you provide the following

- Correct implementation
- Correct specification
- Correct contracts
- Sufficient contracts for verification

Partial credit depending on how many of these you achieve

Clarity & conciseness is necessary for partial credit!

Mini-Projects

Mini-projects are intended to build proficiency in:

- Writing good specifications
- Applying course principles to practice
- Making effective use of automated tools
- **Writing useful & correct code**

Mini-Projects

Mini-projects are intended to build proficiency in:

- Writing good specifications
- Applying course principles to practice
- Making effective use of automated tools
- **Writing useful & correct code**

Gradual progression to sophistication:

1. Familiarize yourself with Why3
2. Implement and prove something
3. Work with more complex data structures
4. Implement and prove something really interesting
5. Optimize your implementation, still verified

Late Policy

Late days

- 5 late days to use throughout the semester
- No more than 2 late days on any assignment
- Late days do not apply to mini-projects!

Logistics

Website: `http://www.cs.cmu.edu/~15414`

Course staff contact: Piazza

Lecture: Tuesdays & Thursdays, 12:20-1:40pm

- First two weeks on ohyay
- Afterwards: HH B131

Lecture Recordings (when remote): YouTube

Office Hours: TBD, schedule on website and course calendar soon

Assignments: Gradescope

Let's prove something!

- Will sort you into breakout rooms
- Figure out mystery function does and how to prove it
- Will recall you to lecture hall
- Let's live-code!