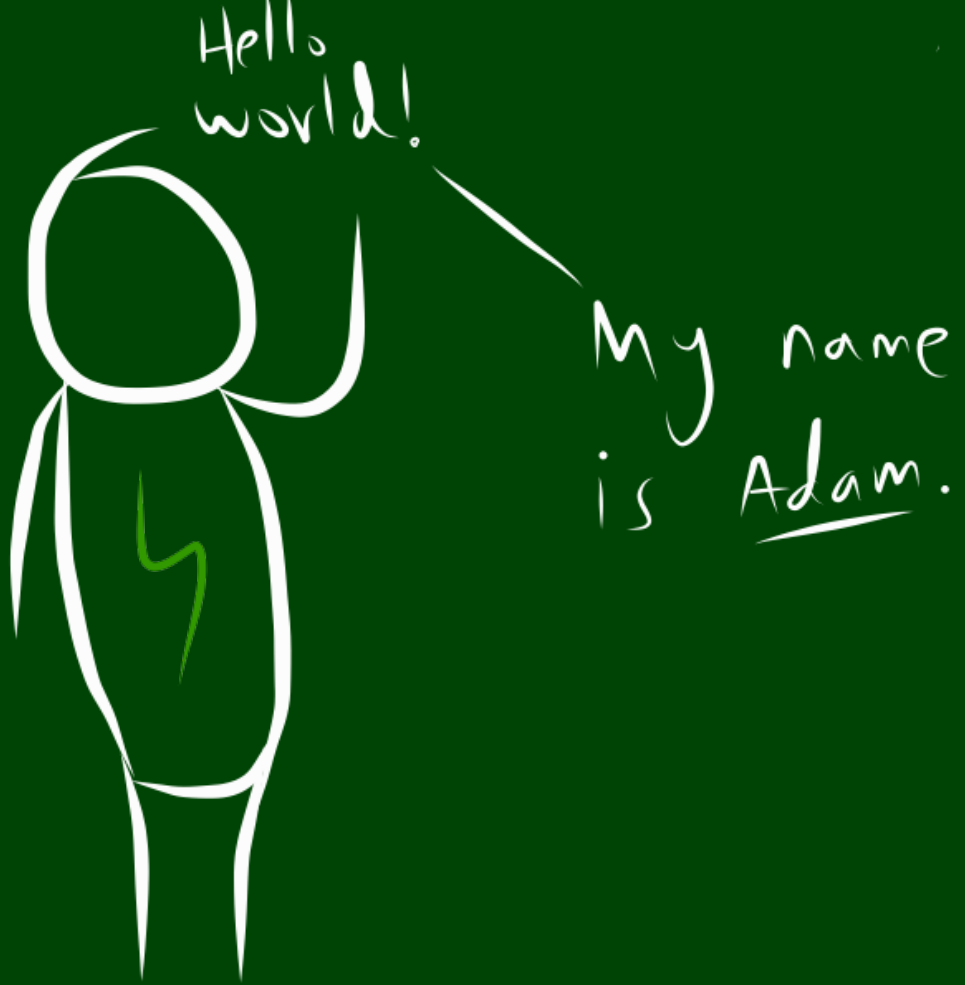
CS 3

Introduction to Software Design

Welcome to CS 3!



Outline

What does it mean for a program to be “correct”?

What does it mean for a program to be "correct"?

What does this code do?

```
_(__,___,____){___/__<=1?_(__,___+1,___ _):!(___%__)?_(__,___+1,0):___
    %__==___ / __&&!____?(printf("%d\t",___/__),_(__,_ __+1,0)):___%__
    >1&&___%__<___/__?_( __,1+ ___,____+!(___/__%(___%__))):___<__*__
    ?_(__,___+1,____):0;}main(){_(100,0,0);}
```

What does it mean for a program to be "correct"?

What does this code do?

```
_(__,___,____){___/__<=1?_(__,___+1,___ _):!(___%__)?_(__,___+1,0):___
    %__==___ / __&&!____?(printf("%d\t",___/__),_(__,_ __+1,0)):___%__
    >1&&___%__<___/__?_( __,1+ ___,____+!(___/__%(___%__))):___<__*__
    ?_(__,___+1,____):0;}main(){_(100,0,0);}
```

Programs must be written for people to read, and only incidentally for machines to execute. (Abelson & Sussman)

What “Program Correctness”?

- The code has the right functionality

What "Program Correctness"?

- The code has the right functionality
- The code is easy to read

What "Program Correctness"?

- The code has the right functionality
- The code is easy to read
- The code is well-documented

What "Program Correctness"?

- The code has the right functionality
- The code is easy to read
- The code is well-documented
- The code is modular

What "Program Correctness"?

- The code has the right functionality
- The code is easy to read
- The code is well-documented
- The code is modular
- The code is easy to re-use

What "Program Correctness"?

- The code has the right functionality
- The code is easy to read
- The code is well-documented
- The code is modular
- The code is easy to re-use
- ...

How Can We Determine If A Program is Correct?

What "Program Correctness"?

- The code has the right functionality
- The code is easy to read
- The code is well-documented
- The code is modular
- The code is easy to re-use
- ...

How Can We Determine If A Program is Correct?

- Write tests

What "Program Correctness"?

- The code has the right functionality
- The code is easy to read
- The code is well-documented
- The code is modular
- The code is easy to re-use
- ...

How Can We Determine If A Program is Correct?

- Write tests
- Think about "edge cases"

What "Program Correctness"?

- The code has the right functionality
- The code is easy to read
- The code is well-documented
- The code is modular
- The code is easy to re-use
- ...

How Can We Determine If A Program is Correct?

- Write tests
- Think about "edge cases"
- Use automated tools to catch unintended issues

What "Program Correctness"?

- The code has the right functionality
- The code is easy to read
- The code is well-documented
- The code is modular
- The code is easy to re-use
- ...

How Can We Determine If A Program is Correct?

- Write tests
- Think about "edge cases"
- Use automated tools to catch unintended issues
- Ask someone experienced to review your code! (more on this later)

Bigger programs lead to bigger problems. . .

Bigger programs lead to bigger problems. . .

- Small programs are simple and malleable
- Big programs are complex and inflexible

Bigger programs lead to bigger problems...

- Small programs are simple and malleable
- Big programs are complex and inflexible

In large programs, interactions become unmanagable

Managing Complexity

Bigger programs lead to bigger problems...

- Small programs are simple and malleable
- Big programs are complex and inflexible

In large programs, interactions become unmanagable

There are a lot of ways to mitigate this, but two of the most useful are:

- Abstraction
- Specification

Abstraction

The essence of abstractions is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context. (John V. Guttag)

Abstraction

The essence of abstractions is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context. (John V. Guttag)

Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts. (Benjamin C. Pierce)

The essence of abstractions is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context. (John V. Guttag)

Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts. (Benjamin C. Pierce)

Some Types of Abstraction

- **Procedural Abstraction.** Splitting a program into functions that each have a single purpose
- **Data Abstraction.** Using ADTs and interfaces to make a boundary between client and implementor

The essence of abstractions is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context. (John V. Guttag)

Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts. (Benjamin C. Pierce)

Some Types of Abstraction

- **Procedural Abstraction.** Splitting a program into functions that each have a single purpose
- **Data Abstraction.** Using ADTs and interfaces to make a boundary between client and implementor

Example

The Caltech CS Introductory Curriculum

Curriculum

Programming	Data Structures	Software Design
Python	Java	C
50 LOC	300 LOC	800 LOC

CS 3 is a practical introduction to designing large programs in a low-level language. Heavy emphasis is placed on documentation, testing, and software architecture. Students will work in teams in two 5-week long projects. In the first half of the course, teams will focus on testing and extensibility. In the second half of the course, teams will use POSIX APIs, as well as their own code from the first five weeks, to develop a large software deliverable. Software engineering topics covered include code reviews, testing and testability, code readability, API design, refactoring, and documentation.

- Lecture on a software topic + team sync-up
- Lab on C and C library skills (solo)
- Project that builds on previous projects (group)
  - first half: physics engine
  - second half: game
- Code review to discuss code quality (alternating with Adam/Sarah & Mentor TAs)

Notable Policies

- Missing more than two weeks of code reviews results in an automatic F in the course.
- To pass the course, you must get at least a "check" on every physics engine project throughout the course.
- You must log all work in the `README` in your repository. Instructions for how to do this are posted on the course website.
- You have been assigned two TA mentors who will work with you throughout the quarter. We've divided course staff into these groups; so, for project help, you should only go to your mentors.

Outline

The most important property of a program is whether it accomplishes the intention of its user. (C. A. R. Hoare)

A specification is a contract between...

The most important property of a program is whether it accomplishes the intention of its user. (C. A. R. Hoare)

A specification is a contract between...

- **Client/Implementor.**
  - The client agrees to only rely on promised information
  - The implementor agrees to always support the promises

The most important property of a program is whether it accomplishes the intention of its user. (C. A. R. Hoare)

A specification is a contract between...

- **Client/Implementor.**
  - The client agrees to only rely on promised information
  - The implementor agrees to always support the promises
- **User/Manufacturer.**
  - The user sets expectations and requirements (no surprises!)
  - The manufacturer doesn't care how it's used (isolation)

The most important property of a program is whether it accomplishes the intention of its user. (C. A. R. Hoare)

A specification is a contract between...

- **Client/Implementor.**
  - The client agrees to only rely on promised information
  - The implementor agrees to always support the promises
- **User/Manufacturer.**
  - The user sets expectations and requirements (no surprises!)
  - The manufacturer doesn't care how it's used (isolation)

Types of Specification

- **Software Specification.** "What game will you be implementing? What are the pieces? When will they be delivered?"
- **Function Specification.** "What are the requirements of this function? What is true after it runs?"

The most important property of a program is whether it accomplishes the intention of its user. (C. A. R. Hoare)

A specification is a contract between...

- **Client/Implementor.**
  - The client agrees to only rely on promised information
  - The implementor agrees to always support the promises
- **User/Manufacturer.**
  - The user sets expectations and requirements (no surprises!)
  - The manufacturer doesn't care how it's used (isolation)

Types of Specification

- **Software Specification.** "What game will you be implementing? What are the pieces? When will they be delivered?"
- **Function Specification.** "What are the requirements of this function? What is true after it runs?"

Right now, we'll focus on function specification.

Precondition

A **precondition** is a predicate that is required for the promises a function makes to happen.

Example Preconditions:

- For `moveRight(int numberOfUnits)`:
- For `minElement(int[] array)`:
- For `add(int index, int value)`:

Preconditions are important, because they explain method behavior to the client.

Precondition

A **precondition** is a predicate that is required for the promises a function makes to happen.

Example Preconditions:

- For `moveRight(int numberOfUnits)`:
  `//@requires numberOfUnits >= 0`
- For `minElement(int[] array)`:
  `//@requires array.length > 0`
- For `add(int index, int value)`:
  `//@requires 0 <= index <= size`

Preconditions are important, because they explain method behavior to the client.

Postcondition

A **postcondition** is a description of behavior that is guaranteed to be true **after a method has run** (if the pre-conditions hold).

Example Postconditions:

- For `moveRight(int numberOfUnits)`:
- For `minElement(int[] array)`:
- For `add(int index, int value)`:

Postconditions are important, because they explain method behavior to the client.

Postcondition

A **postcondition** is a description of behavior that is guaranteed to be true **after a method has run** (if the pre-conditions hold).

Example Postconditions:

- For `moveRight(int numberOfUnits)`:

```
//@ensures Increases the x coordinate of the circle
//         by numberOfUnits
```

- For `minElement(int[] array)`:

```
//@ensures returns the smallest element in array
```

- For `add(int index, int value)`:

```
//@ensures Inserts value at index in the list;
//         shifts all elements from index to the end
//         forward one index
```

Postconditions are important, because they explain method behavior to the client.

Course Specification

```
 1 /**
 2 @requires You know how to program, at the level of CS 2, in
 3           a compiled language
 4 @requires You are interested in learning to write good software,
 5           not just "software that passes the tests"
 6 @requires You are willing and able to work on a team and learn
 7           to be part of a team
 8
 9 @ensures You will have worked on two substantial codebases over
10          a non-trivial period of time
11 @ensures You will have experience learning how to use a new library
12 @ensures You will have experience writing code as part of a team
13 @ensures You will know C at the level necessary to succeed in CS 24
14 @ensures You will know how to debug C at the level necessary to
15          succeed in CS 24
16 */
17 letter course(...) {
18    ...
19 }
```

Outline