

# A note on homework output

Your output should match the spec *exactly*.

- Computers are dumb and don't know what you mean
- We can't anticipate your possible enhancements

# Errors and exceptions

EECS 211

Winter 2017

# Kinds of errors

- Static (compile-time) errors
  - ▶ Syntax errors
  - ▶ Semantic (type) errors
  - ▶ Linker errors
- Dynamic (run-time) errors
  - ▶ Logic errors (bugs)
  - ▶ User and environment errors

# Static versus dynamic errors

Static happens at build time

Dynamic happens at run time

# Static versus dynamic errors

Static happens at build time

Dynamic happens at run time

Consequently, programs with static errors can't be run!

# Syntax errors

When the program doesn't have the correct form for a program.

Examples:

- Unmatches bracket or parenthesis
- Missing or extraneous semicolon
- A reserved word used where an identifier is required

# Semantic errors

When something doesn't make sense. Examples:

- Calling a function that hasn't been declared
- Calling a two-argument function with three arguments
- Using an `int` where a `string` is required

# Linker errors

When some promised definitions are still missing at the end of the build process

(This will make more sense later)



# Logic errors

When the programmer gets something wrong. Examples:

- Integer divide-by-zero
- Array out-of-range error
- Crashes when attempting to render two tables side-by-side

## User and environment errors

When the user does something wrong, or the environment isn't in the required state. Examples:

- Attempting to open a file that doesn't exist
- The network being down
- Clicking in a modally-inactive window

# What should we do in case of error?

It depends:

**Programmer errors** All is lost! So probably crashing is best

**User/env. errors** Be user-friendly! Allow the user to recover

# What should we do in case of error?

It depends:

**Programmer errors** All is lost! So probably crashing is best\*

**User/env. errors** Be user-friendly! Allow the user to recover

\* unless it's required to be robust (like a flight control system)

# What should we do in case of error?

It depends:

**Programmer errors** All is lost! So probably crashing is best\*

**User/env. errors** Be user-friendly! Allow the user to recover†

\* unless it's required to be robust (like a flight control system)

† unless the programmer is the user and the user doesn't care

## Example logic error

```
// Computes the mean value of a vector  
double mean(vector<double> sample)  
{  
    double sum = 0;  
  
    for (double element : sample)  
        sum += element;  
  
    return sum / sample.size();  
}
```

## Example logic error

```
// Computes the mean value of a vector  
double mean(vector<double> sample)  
{  
    double sum = 0;  
  
    for (double element : sample)  
        sum += element;  
  
    return sum / sample.size();  
}
```

Now suppose mean is called with an empty vector...

# Whose job is it to prevent this?

Options:

- The author of mean

(the *service*)



# Whose job is it to prevent this?

Options:

- The author of `mean` (the *service*)
- The author of the code that calls `mean` (the *client*)

# Whose job is it to prevent this?

Options:

- The author of `mean` (the *service*)
- The author of the code that calls `mean` (the *client*)
- Both!

## What the client should do

Try not to call `mean` with an empty vector!

## What the client should do

Try not to call `mean` with an empty vector!

If the empty data set is coming from the user (or a file), the client should present an error message and allow the user to recover

# What the service should do

Several options:

- Just return nonsense
- Crash the program
- Throw an exception
- Declare a precondition (and one of the above)

## Just return nonsense!

```
// Computes the mean value of a vector  
double mean(vector<double> sample)  
{  
    double sum = 0;  
  
    for (double element : sample)  
        sum += element;  
  
    return sum / sample.size();  
}
```

## Just return nonsense!

```
// Computes the mean value of a vector  
double mean(vector<double> sample)  
{  
    double sum = 0;  
  
    for (double element : sample)  
        sum += element;  
  
    return sum / sample.size();  
}
```

### Pros:

- It's fast
- It's simple

### Cons:

- Hard to debug

## Document the precondition and return nonsense

```
// Computes the mean value of a vector  
// PRECONDITION: !sample.empty()  
double mean(vector<double> sample)  
{  
    double sum = 0;  
    for (double element : sample) sum += element;  
    return sum / sample.size();  
}
```



## Document the precondition and return nonsense

```
// Computes the mean value of a vector  
// PRECONDITION: !sample.empty()  
double mean(vector<double> sample)  
{  
    double sum = 0;  
    for (double element : sample) sum += element;  
    return sum / sample.size();  
}
```

### Pros:

- It's fast
- It's simple
- It's clearer

### Cons:

- Still hard to debug

# Crash the program

```
#include <cstdlib>
```

```
double mean(vector<double> sample)
```

```
{
```

```
    if (sample.empty()) {
```

```
        std::cerr << "Empty sample has no mean\n";
```

```
        exit(1);
```

```
    }
```

```
    :
```

```
}
```

# Crash the program

```
#include <cstdlib>
```

```
double mean(vector<double> sample)
```

```
{
```

```
    if (sample.empty()) {
```

```
        std::cerr << "Empty sample has no mean\n";
```

```
        exit(1);
```

```
    }
```

```
    :
```

```
}
```

## Pros:

- Easier to debug
- Still pretty simple

## Cons:

- What if client wants to recover?
- Takes time to check (maybe)

## Throw an exception

```
#include <stdexcept>

double mean(vector<double> sample)
{
    if (sample.empty())
        throw std::runtime_error(
            "empty sample has no mean");
    :
}
```

## Throw an exception

```
#include <stdexcept>

double mean(vector<double> sample)
{
    if (sample.empty())
        throw std::runtime_error(
            "empty sample has no mean");
    :
}
```

# Throw an exception

```
#include <stdexcept>

double mean(vector<double> sample)
{
    if (sample.empty())
        throw std::runtime_error(
            "empty sample has no mean");
    :
}
```

## Pros:

- Easiest to debug
- Allows client to recover

## Cons:

- Takes time to check (maybe)
- More complicated

*To the terminal!*