Handling Erroneous Behavior

Nick Sumner
wsumner@sfu.ca
Sources of Error

- Your software exists in an adversarial context
Sources of Error

- Your software exists in an adversarial context
  - Users (both ignorant & malign)
Sources of Error

- Your software exists in an adversarial context
  - Users (both ignorant & malign)
  - External software components
Sources of Error

- Your software exists in an adversarial context
  - Users (both ignorant & malign)
  - External software components
  - Internal software components
Sources of Error

- Your software exists in an adversarial context
  - Users (both ignorant & malign)
  - External software components
  - Internal software components
  - Environmental context
Sources of Error

- Your software exists in an adversarial context
  - Users (both ignorant & malign)
  - External software components
  - Internal software components
  - Environmental context

- You should develop your software to respond appropriately to erroneous behavior
Sources of Error

- Your software exists in an adversarial context
  - Users (both ignorant & malign)
  - External software components
  - Internal software components
  - Environmental context

- You should develop your software to respond appropriately to erroneous behavior
  - The challenge is knowing what to do & when
User Error

- The user is an adversary
User Error

- The user is an adversary
  - If they can do the wrong thing, they will
User Error

- The user is an adversary
  - If they can do the wrong thing, they will
  - If they can benefit from it, they will seek to
User Error

- The user is an adversary
  - If they can do the wrong thing, they will
  - If they can benefit from it, they will seek to

Mallory, how much money would you like to transfer to Bob?
**User Error**

- The user is an adversary
  - If they can do the wrong thing, they will
  - If they can benefit from it, they will seek to

Mallory, how much money would you like to transfer to Bob? $500.00
User Error

- The user is an adversary
  - If they can do the wrong thing, they will
  - If they can benefit from it, they will seek to

Mallory, how much money would you like to transfer to Bob?
$-500.00
User Error

- The user is an adversary
  - If they can do the wrong thing, they will
  - If they can benefit from it, they will seek to

Mallory, how much money would you like to transfer to Bob?
$-500.00

Ask yourself what should be allowable & enforce it
User Error

- The user is an adversary
  - If they can do the wrong thing, they will
  - If they can benefit from it, they will seek to

- Validate & sanitize all user input
  - Command line
  - Files
  - Databases
  - ...
User Error

- The user is an adversary
  - If they can do the wrong thing, they will
  - If they can benefit from it, they will seek to

- Validate & sanitize all user input
  - Command line
  - Files
  - Databases
  - ...

- Prefer to provide feedback indicating the user error
User Error

- The user is an adversary
  - If they can do the wrong thing, they will
  - If they can benefit from it, they will seek to

- Validate & sanitize all user input
  - Command line
  - Files
  - Databases
  - ...

- Prefer to provide feedback indicating the user error

- You can even use software hardening tools for better security (more in CMPT 473)
Handling Non-user Errors

- What if a function returns an unexpected value?
  - Can’t just print an error message for that function and ask it to return again....
Handling Non-user Errors

- What if a function returns an unexpected value?
  - Can’t just print an error message for that function and ask it to return again....

- Strategies for erroneous scenarios
  - Design them out of existence

Similar to what we did with ambiguous function arguments.
Handling Non-user Errors

- What if a function returns an unexpected value?
  - Can’t just print an error message for that function and ask it to return again....

- Strategies for erroneous scenarios
  - Design them out of existence
  - Assertions
Handling Non-user Errors

• What if a function returns an unexpected value?
  – Can’t just print an error message for that function and ask it to return again....

• Strategies for erroneous scenarios
  – Design them out of existence
  – Assertions
  – Exceptions
Handling Non-user Errors

- What if a function returns an unexpected value?
  - Can’t just print an error message for that function and ask it to return again....

- Strategies for erroneous scenarios
  - Design them out of existence
  - Assertions
  - Exceptions
  - Return error codes & out arguments
Handling Non-user Errors

- What if a function returns an unexpected value?
  - Can’t just print an error message for that function and ask it to return again.

- Strategies for erroneous scenarios
  - Design them out of existence
  - Assertions
  - Exceptions
  - Return error codes & out arguments

- All of these come with a cost and trade one form of complexity for another.
Defining Away Erroneous Behavior

- Use the type system to your advantage

```python
computeForce(Mass{16g}, Acceleration{9.8mss})
```
Defining Away Erroneous Behavior

- Use the type system to your advantage
- Generalize away corner cases
Defining Away Erroneous Behavior

- Use the type system to your advantage
- Generalize away corner cases
  - Implicitly – e.g. Null Object Pattern

**Null Object Pattern**

Create a subtype representing an object with no information.

Any getters/methods effectively perform no-ops.
Defining Away Erroneous Behavior

- Use the type system to your advantage

- Generalize away corner cases
  - Implicitly – e.g. Null Object Pattern
  - Explicitly – e.g. getChildren() vs getLeft() & getRight()

What are the trade offs?
Defining Away Erroneous Behavior

- Use the type system to your advantage
- Generalize away corner cases
- Make inconsistent state unrepresentable
  - State Pattern – richer state machines
  - Sum types – e.g. boost::variant & std::variant (& optional!)
Defining Away Erroneous Behavior

- Use the type system to your advantage
- Generalize away corner cases
- Make inconsistent state unrepresentable
Defining Away Erroneous Behavior

- Use the type system to your advantage
- Generalize away corner cases
- Make inconsistent state unrepresentable

```cpp
enum class CurrentState { SLEEP, PLAY, WORK }

class Student { 
    CurrentState state; 
    uint64_t timeWorked; 
};
```
Defining Away Erroneous Behavior

- Use the type system to your advantage
- Generalize away corner cases
- Make inconsistent state unrepresentable

```cpp
enum class CurrentState { SLEEP, PLAY, WORK };

class Student {
    CurrentState state;
    uint64_t timeWorked;
}
```

What can go wrong?
State Patterns & Sum Types

- How can we fix it?
State Patterns & Sum Types

- How can we fix it?

class CurrentState {
  ...
};
State Patterns & Sum Types

- How can we fix it?

class CurrentState {
    ...
};

class Sleep : public CurrentState {
};

class Work : public CurrentState {
    uint64_t timeWorked
};
State Patterns & Sum Types

• How can we fix it?

class Student {
  unique_ptr<CurrentState> state;
};

class CurrentState {
  ...
};

class Sleep : public CurrentState {
};

class Work : public CurrentState {
  uint64_t timeWorked
};
State Patterns & Sum Types

- How can we fix it?

```cpp
class Student {
    unique_ptr<CurrentState> state;
};

class CurrentState {
    // This is part of the state pattern!
};

class Sleep : public CurrentState {
};

class Work : public CurrentState {
    uint64_t timeWorked
};
```
State Patterns & Sum Types

• How can we fix it?

class Student {
    struct Sleep {}
    struct Play {}
    struct Work { uint64_t timeWorked; }

    boost::variant<Sleep,
                    Play,
                    Work> currentState;
};

This uses sum types!
Defining Away Erroneous Behavior

- Use the type system to your advantage
- Generalize away corner cases
- Make inconsistent state unrepresentable
  - State Pattern – richer state machines
  - Sum types – e.g. boost::variant & std::variant (& optional!)
Defining Away Erroneous Behavior

- Use the type system to your advantage
- Generalize away corner cases
- Make inconsistent state unrepresentable
  - State Pattern – richer state machines
  - Sum types – e.g. boost::variant & std::variant (& optional!)

---

State Pattern – richer state machines
Sum types – e.g. boost::variant & std::variant (& optional!)
Defining Away Erroneous Behavior

- Use the type system to your advantage
- Generalize away corner cases
- Make inconsistent state unrepresentable
  - State Pattern – richer state machines
  - Sum types – e.g. boost::variant & std::variant (& optional!)

```cpp
boost::optional<int>
divide(int numerator, int denominator);
```
Defining Away Erroneous Behavior

- Use the type system to your advantage
- Generalize away corner cases
- Make inconsistent state unrepresentable
  - State Pattern – richer state machines
  - Sum types – e.g. boost::variant & std::variant (& optional!)
  - Phantom Types – Exploit parametric polymorphism
double distanceTraveled(double speed, double time) {
    return speed * time;
}

What can go wrong?
Phantom Types

double
distanceTraveled(double speed, double time) {
    return speed * time;
}

// Miles per
... = distanceTraveled(3, 5);

d1 = ...; // Meters
d2 = ...; // Miles
... = d1 + d2; // Uh oh.

What can go wrong?
Phantom Types

- Parameterize your types by unique type names...

```cpp
struct Meters {};  
struct Miles {};   
struct Seconds {};  
struct Hours {};    

template<typename T, typename U>  
struct Speed { double speed; };    

template<typename T>  
struct Distance { double distance; };  

template<typename T>  
struct Time { double time; };    
```
Phantom Types

- Consistent units are enforced via template arguments

```cpp
template <typename T, typename U>
Distance<T> distanceTraveled(Speed<T,U> speed, Time<U> time) {
    return {speed.speed * time.time};
}

template <typename T>
Distance<T> operator+(Distance<T> d1, Distance<T> d2) {
    return d1.distance + d2.distance;
}
```
Phantom Types

distanceTraveled(Speed<Miles, Hours>{3},
    Time<Seconds>{5});

phantom.cpp:37:19: error: no matching function for call to 'distanceTraveled'
... deduced conflicting types for parameter 'U' ('Hours' vs. 'Seconds')
 Phantom Types

distanceTraveled(Speed<\text{Miles},\text{Hours}>{3},
\text{Time<Seconds}>{5});

phantom.cpp:37:19: error: no matching function for call to 'distanceTraveled'
... deduced conflicting types for parameter 'U' ('\text{Hours}' vs. '\text{Seconds}')

\begin{verbatim}
d1 = distanceTraveled(Speed<\text{Miles},\text{Hours}>{3},
                     \text{Time<Hours}>{5});
d2 = distanceTraveled(Speed<\text{Meters},\text{Seconds}>{3},
                     \text{Time<Seconds}>{5});
d3 = distance2 + distance3;
\end{verbatim}

phantom.cpp:41:30: error: invalid operands to binary expression
... deduced conflicting types for parameter 'T' ('\text{Miles}' vs. '\text{Meters}')
Phantom Types

distanceTraveled(Speed<Miles, Hours>{3},
    Time<Seconds>{5});

phantom.cpp:37:19: error: no matching function for call to 'distanceTraveled'
... deduced conflicting types for parameter 'U' ('Hours' vs. 'Seconds')

d1 = distanceTraveled(Speed<Miles, Hours>{3},
    Time<Hours>{5});

d2 = distanceTraveled(Speed<Meters, Seconds>{3},
    Time<Seconds>{5});

d3 = distance2 + distance3;

phantom.cpp:41:30: error: invalid operands to binary expression
... deduced conflicting types for parameter 'T' ('Miles' vs. 'Meters')

What are the trade offs for using this technique?
Assertions

- Assertions check the **invariants** of your program
Assertions

- Assertions check the *invariants* of your program
  - What should be true when a function starts?
  - What should be true when a function ends?
Assertions

• Assertions check the invariants of your program
  – What should be true when a function starts?
  – What should be true when a function ends?

• These are guaranteed bugs that should never happen in production!
Assertions

- Assertions check the invariants of your program
  - What should be true when a function starts?
  - What should be true when a function ends?

- These are guaranteed bugs that should never happen in production!

```c
#include <cassert>

constexpr char ascii[256] = ...

char getChar(int asciiCode) {
    assert(0 < asciiCode && asciiCode < 256 && "ASCII code out of range."");
}
```
Assertions

- Assertions check the invariants of your program
  - What should be true when a function starts?
  - What should be true when a function ends?

- These are guaranteed bugs that should never happen in production!

- In general, more assertions leads to better quality code.
Exceptions

- Exceptions respond to *external* unexpected behaviors.
Exceptions

- Exceptions respond to *external* unexpected behaviors.

- What should you do when an exception is thrown?
Exceptions

- Exceptions respond to *external* unexpected behaviors.

- **What should you do when an exception is thrown?**
  - Nothing?
  - Try again?
  - Log the error & continue?
  - Log the error & abort?
Exceptions

● Exceptions respond to *external* unexpected behaviors.

● What should you do when an exception is thrown?
  - Nothing?
  - Try again?
  - Log the error & continue?
  - Log the error & abort?

● What should you pass to an exception when throwing?
Exceptions

- Exceptions respond to *external* unexpected behaviors.

- What should you do when an exception is thrown?
  - Nothing?
  - Try again?
  - Log the error & continue?
  - Log the error & abort?

- What should you pass to an exception when throwing?
  - Do you expect it to be re-trying?
  - Do you expect it to be logged?
Handling Erroneous Behavior

- As a developer, how do you respond to erroneous behavior?
Handling Erroneous Behavior

- As a developer, how do you respond to erroneous behavior?

  What if the cause occurred much earlier?
Handling Erroneous Behavior

- As a developer, how do you respond to erroneous behavior?

- What if an *absence* of behavior is erroneous?
Handling Erroneous Behavior

- As a developer, how do you respond to erroneous behavior?
- What if an absence of behavior is erroneous?
- What if a trend makes something erroneous?
Handling Erroneous Behavior

- As a developer, how do you respond to erroneous behavior?
- What if an absence of behavior is erroneous?
- What if a trend makes something erroneous?
- What if it only happens when deployed?
Handling Erroneous Behavior

- As a developer, how do you respond to erroneous behavior?
- What if an absence of behavior is erroneous?
- What if a trend makes something erroneous?
- What if it only happens when deployed?

*Tracking* behavior is crucial. Real world software uses *logging*. 
Logging

- A logging system records program state & events over time.
Logging

• A logging system records program state & events over time.

```cpp
LOG(INFO) << "Creating new account. "
<< "name:" << username;
```
A logging system records program state & events over time.

```cpp
LOG(INFO) << "Creating new account. "
   << "name:" << username;
```
Loggers are systems that record program state and events over time.

```plaintext
LOG(INFO) "Creating new account. " "name:" username;
```
Logging

- A logging system records program state & events over time.

```cpp
LOG(INFO) << "Creating new account. "
  << "name:" << username;

LOG_IF(INFO, numUsers > 10)
  << "Many users logged in. "
  << "numusers:" << numUsers;
```
Logging

- A logging system records program state & events over time.

```cpp
LOG(INFO) << "Creating new account. "
      << "name:" << username;

LOG_IF(INFO, numUsers > 10)
      << "Many users logged in. "
      << "numusers:" << numUsers;

CHECK_LT(index, size) << "Index out of bounds."
CHECK_NOTNULL(ptr);
```
Logging

- A logging system records program state & events over time.

- **Common to log:** [Fu et al., ICSE 2014]
Logging

- A logging system records program state & events over time.

- **Common to log:** [Fu et al., ICSE 2014]
  - Assertion failures
  - Critical return values
  - Exceptions
  \[\text{Unexpected Situations}\]
Logging

• A logging system records program state & events over time.

• Common to log: [Fu et al., ICSE 2014]
  
  - Assertion failures
  - Critical return values
  - Exceptions
  - Key branch points
  - Observation points

  \( \text{Key Execution Points} \)

  \( \text{Unexpected Situations} \)
Logging

- A logging system records program state & events over time.

- **Common to log:** [Fu et al., ICSE 2014]
  - Assertion failures
  - Critical return values
  - Exceptions
  - Key branch points
  - Observation points

- Logging *too little* or *too much* can be a problem
Logging

- A logging system records program state & events over time.

- **Common to log:** [Fu et al., ICSE 2014]
  - Assertion failures
  - Critical return values
  - Exceptions
  - Key branch points
  - Observation points

- **Logging too little or too much** can be a problem
  - Might miss what you want
  - Might create a haystack for your needle
  - Might spend too many resources!
Logging Guidelines

- Log all assertion failures
Logging Guidelines

- Log all assertion failures
- Log exceptions at most once
Logging Guidelines

- Log all assertion failures

- Log exceptions at most once
  - Might *defer* logging if exception is rethrown
Logging Guidelines

- Log all assertion failures

- Log exceptions at most once
  - Might *defer* logging if exception is rethrown
  - Might *skip* logging exceptions that do no harm (e.g. if deleting a file failed because it was not there)
Logging Guidelines

- Log all assertion failures
- Log exceptions at most once
  - Might *defer* logging if exception is rethrown
  - Might *skip* logging exceptions that do no harm (e.g. if deleting a file failed because it was not there)
- Log all events needed for auditing
Logging Guidelines

- Log all assertion failures
- Log exceptions at most once
  - Might *defer* logging if exception is rethrown
  - Might *skip* logging exceptions that do no harm
    (e.g. if deleting a file failed because it was not there)
- Log all events needed for auditing
- **Log logic that provides context for possible errors**
Logging Guidelines

- Log all assertion failures
- Log exceptions at most once
  - Might *defer* logging if exception is rethrown
  - Might *skip* logging exceptions that do no harm (e.g. if deleting a file failed because it was not there)
- Log all events needed for auditing
- Log logic that provides context for possible errors

Bear in mind, logging also comes at a price. It is a *cross-cutting concern*. 
Logging Guidelines

- Make your log easy to use
  - Machine parsable if possible
Logging Guidelines

- Make your log easy to use
  - Machine parsable if possible
  - What / When / Why / Where should be clearly captured
Summary

- Many strategies for dealing with possible errors.
Summary

- Many strategies for dealing with possible errors.
- Designing them away is preferred.
Summary

- Many strategies for dealing with possible errors.
- Designing them away is preferred.
- All strategies have a cost.
Summary

- Many strategies for dealing with possible errors.
- Designing them away is preferred.
- All strategies have a cost.
- Logging is critical for dealing with real world code.