Engineering Robust Server Software

Exceptions
Exceptions

- Handling problems: exceptions
- C++
  - temp-and-swap
  - RAII
  - Smart Pointers
- Java
  - finally
  - specifications
  - finalizers (and why they are not what you need for this)
Exceptions

- Review: exceptions = way to handle problems
  - Thing goes wrong? throw exception
  - Know how to deal with problem? try/catch exception
    - In python, try/except

- Why exceptions?
  - Return error code? Cluttery, easy to forget/ignore
  - Do nothing? Automatically pass problem to caller
  - Provide details about error
Exceptions: Downsides

- So exceptions: best idea ever?
- Downsides too
  - Unexpected things happen in code
    - Well, that is true anyways
  - Used improperly: corrupted objects, resource leaks, ...
- Bottom line:
  - Good if you do all things right
### Exception Safety

- Continued review: exception safety
  - Remind us of the four levels of exceptions safety?

<table>
<thead>
<tr>
<th>Stronger Guarantees</th>
<th>No Throw</th>
<th>Strong</th>
<th>Basic</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Will not throw any exception. Catches and handles any exceptions throw by operations it uses</td>
<td>No side-effects if an exception is thrown: objects are unmodified, and no memory is leaked</td>
<td>Objects remain in valid states: no dangling pointers, invariants remain intact. No memory is leaked</td>
<td>Does not provide even a basic exception guarantee. Unacceptable in professional code.</td>
</tr>
</tbody>
</table>
template<typename T>
class LList {
  // other things omitted, but typical
  LList & operator=(const LList & rhs) {
    if (this != &rhs) {
      deleteAll();
      Node * curr = rhs.head;
      while (curr != null) {
        addToBack(curr->data);
        curr = curr->next;
      }
    }
    return *this;
  }
};

Which guarantee does this make?
A: Strong
B: Basic
C: No Guarantee
D: Need more info...
template<typename T>
class LList {
    // other things omitted, but typical
    LList & operator=(const LList & rhs) {
        if (this != &rhs) {
            deleteAll();
            Node * curr = rhs.head;
            while (curr != null) {
                addToBack(curr->data);
                curr = curr->next;
            }
        }
        return *this;
    }
};
template<typename T>
class LList {
    //other things omitted, but typical

    void deleteAll() {
        while (head != nullptr) {
            Node * temp = head->next;
            delete head;
            head = temp;
        }
        tail = nullptr;
    }
};
template<typename T>
class LList {
    // other things omitted, but typical

    void addToBack(const T& d) {
        Node * newNode = new Node(d, nullptr, tail);
        if (tail == nullptr) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }
};

Which guarantee does addToBack() make?

Depends on copy constructor for T

Could throw memory allocation exception, but does so before any changes

<table>
<thead>
<tr>
<th>T's Copy Constructor</th>
<th>addToBack()</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Throw</td>
<td>Strong</td>
</tr>
<tr>
<td>Strong</td>
<td>Strong</td>
</tr>
<tr>
<td>Basic</td>
<td>Basic</td>
</tr>
<tr>
<td>No Guarantee</td>
<td>No Guarantee</td>
</tr>
</tbody>
</table>
template<typename T>
class LList {
    // other things omitted, but typical
    LList & operator=(const LList & rhs) {
        if (this != &rhs) {
            // no throw
            deleteAll(); // no throw
            Node * curr = rhs.head; // no throw
            while (curr != null) {
                addToBack(curr->data); // strong [let us suppose]
                curr = curr->next; // no throw
            }
        }
        return *this; // no throw
    }
};
LList & operator=(const LList & rhs) {
    if (this != &rhs) {
        Node * temp = rhs.head;
        Node * n1 = nullptr;
        Node * n2 = nullptr;
        if (temp != nullptr){
            n1 = n2 = new Node(temp->data, nullptr, nullptr);
            temp = temp->next;
            while (temp != nullptr) {
                n2->next = new Node(temp->data, n2, nullptr);
                n2 = n2->next;
                temp = temp->next;
            }
        } deleteAll();
        head = n1; tail = n2;
    } return *this;
}

Which guarantee does this version make?

No guarantee! :(

Why?
If we have exception while building the new list, then that memory is lost and therefore leaked.
template<typename T>
class LList {
    //other things omitted, but typical
    LList & operator=(const LList & rhs) {
        if (this != &rhs) {
            LList temp(rhs);  // Stack allocated!
            std::swap(temp.head, head);
            std::swap(temp.tail, tail);
        }
        return *this;
    }
};

A good strategy to improve exception safety: **temp-and-swap**
(also called **copy-and-swap**)

Which guarantee does this make?

**Strong!**

Why?
Only place we can get an exception is when making temp (no changes made yet); swap is no-throw.
Temp-and-swap

- Common idiom for strong guarantees: temp-and-swap
  - Make `temp` object
  - Modify `temp` object to be what you want `this` to be
  - swap fields of `temp` and `this`
  - `temp` destroyed when you return (destructor cleans up state)
    - Exception? `temp` destroyed in stack unwinding
- Downside?
  - Change only some state: may be expensive to copy entire object
What About This Code...

template<typename T>
class LList {
    //other things omitted, but typical
    LList & operator=(const LList & rhs) {
        if (this != &rhs) {
            m.lock();
            rhs.m.lock();
            LList temp(rhs);  //What if this throws?
            std::swap(temp.head, head);
            std::swap(temp.tail, tail);
            rhs.m.unlock();
            m.unlock();
        }
        return *this;
    }
};

We've acquired locks and are not releasing them!
template<typename T>
class LList {
  //other things omitted, but typical
  LList & operator=(const LList & rhs) {
    if (this != &rhs) {
      std::lock_guard<std::mutex> lck1(m);
      std::lock_guard<std::mutex> lck2(rhs.m);
      LList temp(rhs);
      std::swap(temp.head, head); //calls m.lock()
      std::swap(temp.tail, tail); //calls rhs.m.lock()
    }
    return *this;
  }
};
template<typename T>
class LList {
    // other things omitted, but typical
    LList & operator=(const LList & rhs) {
        if (this != &rhs) {
            std::lock_guard<std::mutex> lck1(m);
            std::lock_guard<std::mutex> lck2(rhs.m);
            LList temp(rhs);
            A: std::swap(temp.head, head);
            std::swap(temp.tail, tail);
            B: }
        C: return *this;
    }
};

Where are these locks unlocked?

A: They are not unlocked anywhere

B: Locks are auto-released here (either naturally or on exception)

C++
RAII

- Resource Acquisition Is Initialization
  - Resource lifetime tied to object lifetime
  - Allocation during initialization
  - Released during destruction

- Example resources:
  - Mutex: lock/unlock
  - Heap Memory: new/delete
  - File: open/close

- Exception safety benefits?

Release-on-destruction means we release resources whether we go down the normal execution path or exception path.
RAII with Heap Objects

- RAII can be applied to Heap Objects via **Smart Pointers**
  - Objects that wrap pointer and provide RAII
  - C++03: `std::auto_ptr` (deprecated)
- C++11:
  - `std::unique_ptr`
  - `std::shared_ptr`
  - `std::weak_ptr`
std::unique_ptr

{
    std::unique_ptr<Thing> thing1 (new Thing);
    //other code here
}
//thing1 goes out of scope: delete its pointer

- Owns a pointer
  - When destroyed, deletes owned pointer
std::unique_ptr

```cpp
{
    std::unique_ptr<Thing> thing1 (new Thing);
    // other code here
    Thing * tp = thing1.get();
}
```

- Owns a pointer
  - When destroyed, deletes owned pointer
- Can use .get() to get raw pointer
std::unique_ptr

{
    std::unique_ptr<Thing> thing1 (new Thing);
    //other code here
    Thing * tp = thing1.get();
    thing1->doSomething();
}

. Owns a pointer
   . When destroyed, deletes owned pointer
. Can use .get() to get raw pointer
. Can also use * and -> operators
std::unique_ptr

{
    std::unique_ptr<Thing> thing1 (new Thing);
    //… … …
    std::unique_ptr<Thing> thing2 (thing1);

    //thing2 owns pointer, thing1 is empty (holds nullptr)
}

- Assignment operator/copy constructor transfer ownership
Thing * foo(int x, char c) {
    Widget * w = new Widget(x);
    Gadget * g = new Gadget(c);
    Thing * t = new Thing(w, g);
    return t;
}

Which guarantee does this make?
A: No Throw
B: Strong
C: Basic
D: No guarantee

Why?
If Thing throws exception, then w and g are leaked!
Exception Safety

```cpp
Thing * foo(int x, char c) {    
    std::unique_ptr<Widget> w (new Widget(x));
    std::unique_ptr<Gadget> g (new Gadget(c));
    Thing * t = new Thing(w.get(), g.get());
    return t;
} // w and g go out of scope here, so... what happens to their pointers?

Is this code correct?
A: Yes
B: No
C: I'm lost on unique_ptr
```

**Why?**
We pass in the `underlying` pointers to Thing, but `unique_ptr` will delete those when this function exits scope.

Thing ends up with two dangling pointers 😞
Exception Safety

```cpp
Thing * foo(int x, char c) { 
    std::unique_ptr<Widget> w (new Widget(x));
    std::unique_ptr<Gadget> g (new Gadget(c));
    Thing * t = new Thing(w.release(), g.release());
    return t;
}
```

`release` returns the pointer (like `get`),
but also gives up ownership (sets the owned pointer to `nullptr`)

Is this code correct?
A: Yes
B: No

Why?
What if `Thing()` constructor throws exception?
We still had to run `release()` first to call it (`probably`).*
Result: more leaked pointers 😞

* Actually unspecified in the C++ standard.
Exception Safety

```cpp
Thing * foo(int x, char c) {
    std::unique_ptr<Widget> w (new Widget(x));
    std::unique_ptr<Gadget> g (new Gadget(c));
    Thing * t = new Thing(w, g);
    return t;
}
```

First: What am I assuming Thing's constructor takes now?
A: Thing (std::unique_ptr<Widget> & , std::unique_ptr<Gadget> &)
B: Thing(Widget * , Gadget *)
C: Thing(Widget, Gadget)
D: Thing (const Widget & , const Gadget &)

Second: Is this code now correct?
A: Yes
B: No
Shared Pointers + Weak Pointers

- **Unique Pointers**: exactly one owner
  - Assignment **transfers** ownership

- **Shared Pointers**: many owners
  - Copying increments count of owners
  - Destruction decrements counts of owners
  - Object freed when owner count reaches 0

- **Weak Pointers**: non-owners of shared pointer
  - Can reference object, but does not figure into owner count
  - Use .lock() to obtain shared_ptr: has object (if exists) or nullptr (if not)
Real C++: Use RAII

- You learned C++ from C
  - We did a lot of things to transition gently
  - Looked somewhat C-like
    - Less C-like and more C++-like as we progressed
- Real C++:
  - Use RAII for everything
Java Exceptions: Slightly Different

- RAII: C++, but not Java
  - Why not in Java? No objects in stack in Java (all in heap...)
- Java's plan: finally
  - **ALWAYS** executed, no matter whether exception or not
Java Exceptions: Slightly Different

```java
public void doAThing(String name) {
    SomeResource sr = null;
    try {
        sr = new SomeResource(name);
        doStuff(sr);
    }
    catch(WhateverException we) {
        dealWithProblem(we);
    }
    finally {
        if(sr != null) {
            sr.close();
        }
    }
}
```
Java Exceptions: Slightly Different

```java
public void doAThing(String name) throws WhateverException {
    SomeResource sr = null;
    try {
        sr = new SomeResource(name);
        doStuff(sr);
    }
    finally {
        if (sr != null) {
            sr.close();
        }
    }
}
```

Can have try-finally (no catch)
- Allows exception to propagate out
- Cleans up resources
Java Exceptions: Slightly Different

```java
public void doAThing(String name) throws WhateverException{
    try (SomeResource sr = new SomeResource(name)) {
        doStuff(sr);
    }
}
```

Java also has try-with-resource
* declare/initialize AutoCloseable object in () after try
  - can have multiple declarations, separate with ;
* automatically makes a finally which closes it
  - closes in reverse order of creation
* can have explicit catch or finally if you want
Java Exceptions: Slightly Different

```java
public void doAThing(String name) throws WhateverException{
    SomeResource sr = null;
    try {
        sr = new SomeResource(name);
        doStuff(sr);
    } finally {
        if(sr != null) {
            sr.close();
        }
    }
}
```

Java's exception specification rules different from C++'s
Exception Specifications

- C++ 03
  - No declaration: can throw anything
  - Declaration: restricted to those types `throw(x, y, z)` or `throw()`
    - Checked at runtime: when exception is thrown
    - If lied, `std::unexpected()`
Exception Specifications

C++ 11

- C++03 specifications valid but deprecated
- noexcept for "no throw"
  - Can take a **boolean expression** to indicate behavior (true=noexcept)
  - **noexcept(expr)** queries if expr is declared noexcept
- If noexcept actually throws, calls std::terminate()
Java

- Two types of exceptions: checked and unchecked
  - **Checked**: exception specifications checked at compile time
    - Compiler ensures you don't lie (aka miss one)
  - **Unchecked**: no need to declare in spec
    - Possible in too many places, would clutter code

---

**Exception Specifications**

- **Throwable**
  - **Error**
  - **Exception**
    - **IOException**
    - **SQLException**
    - **RuntimeException**
Exception Specifications

Too rare:
"Reasonable" applications do not try/catch these

Java
Exception Specifications

Too common: 
RuntimeException: too ubiquitous to clutter code with specifications (everything might throw them)

Java

ArrayIndexOutOfBoundsException
NullPointerException
ArithmeticException

......

NullPointerException

Exception

IOException
SQLException

class Throwable

Error

IOException
SQLException

NullPointerException
ArithmeticException

......

ArrayIndexOutOfBoundsException

NullPointerException

ArithmeticException
Exception Specifications

Checked exceptions:
- Rare enough to merit specification
- Reasonable enough to try/catch

[Diagram showing the hierarchy of exceptions]

Java
Java: Finalizers

- Java objects have `.finalize()`
  - "Called by the garbage collector on an object when garbage collection determines that there are no more references to the object."
- Seems like maybe we could use this to help resource management?
Let's Look at Stack Overflow

When the IO resource is an instance variable, then you should close it in the `finalize()` method.

The `finalize()` method is called by the Java virtual machine (JVM) before the program exits to give the program a chance to clean up and release resources. Multi-threaded programs should close all files and sockets they use before exiting so they do not face resource starvation. The call to `server.close()` in the `finalize()` method closes the Socket connection used by each thread in the program.

```java
protected void finalize(){
    //Objects created in run method are finalized when
    //program terminates and thread exits
    try{
        server.close();
    } catch (IOException e) {
        System.out.println("Could not close socket");
        System.exit(-1);
    }
}
```

Finalizer: NOT For Resource Management

- Do NOT try to use finalizers for resource management!
  - No guarantee of when they will run (may never gc object!)
- Do NOT use finalizers in general
  - May run on other threads (possibly multiple finalizers at once)
    - Were you thinking about how to synchronize them?
    - What about deadlock?
  - Likely to run when memory is scarce (may cause problems if you allocate)
  - Could accidentally make object re-referenceable?
Exceptions

- Handling problems: exceptions
- C++
  - temp-and-swap
  - RAII
  - Smart Pointers
- Java
  - finally
  - specifications
  - finalizers (and why they are not what you need for this)