COMP6771 19T2 (https://webcms3.cse.unsw.edu.au/COMP6771/19T2)

Week 06 Tutorial Sample Answers

Advanced C++ Programming (https://webcms3.cse.unsw.edu.au/COMP6771/19T2)

1. What are examples of commonly used C++ exception types? When would we use them?

    - **std::invalid_argument**:
    - **std::out_of_bounds**:

2. What's an example of a case we'd want to rethrow an exception?

    - When you aren't interested in managing a caught exception (wanting to leave that to a higher level), but you would like to log something to do with the exception

3. In what cases would we want to have two catch statements for a thrown exception?

    When we have some type-specific exceptions to catch (e.g. std::out_of_range), but then also want to handle other exceptions (known or unknown) in other cases (e.g. std::length_error, std::exception).

4. When would we use a shared pointer instead of a unique pointer?

    When we are unsure which of a series of pointers (to a single resource) will have the longest lifetime. If it's known which will outlast the others, then we use a unique pointer with a number of observers.

5. What is **stack unwinding**?

    The process of exiting the stack frames until we find an exception handler for the function.

    What happens during stuck unwinding?

    Destructors are called on relevant objects on the way out; and any exceptions thrown in a destructor (that are not caught and handled in that destructor) cause std::terminate to be called

    What issue can this potentially cause? How can we fix this?

    If unmanaged resources were created before an exception throw, they may not be appropriately released. The solution is to ensure that every resource is owned by either another resource, the stack, or an unnamed temporary variable.

6. What are the 4 exception safety levels and what does each mean?

    - No-throw (failure transparency): Operations guaranteed to succeed
    - Strong exception safety (commit-or-rollback): Operations can fail, but failed operations will have no side effects
    - Weak exception safety (no-leak): May cause side effects, but guaranteed that no memory leaks will occur
    - No exception safety: No guarantees

7. Consider the following code:

```
#include <string>
#include <stdexcept>

class NamedIntPointer {
  NamedIntPointer::NamedIntPointer(int value, const std::string& name):
      ptr_{new int{value}}, name_{name} {
    if (name.empty()) {
      throw std::invalid_argument{"The int pointer must be named"};
    }
  }

  ~NamedIntPointer() {
    delete ptr_;
  }

  int* ptr_;
  std::string name_;
}
```

Find the bug, and discuss what we could do to solve the issue, including the upsides and downsides of each potential solution.

The bug is that the object is partially constructed when an exception is thrown (since the constructor doesn't get to finish), and as such, the destructor will not be called if the name is empty.

Potential solutions:

- Use a unique pointer. While the object will not be destructed, any fully constructed subobjects (like a smart pointer) will be, so this will solve our issue. It's simple and clean
- Ensure that any potential exceptions are thrown before we obtain resources by putting the new after the throw (initialising ptr outside of the initialiser list). This is considered bad style.
- Ensure that any potential exceptions are thrown before we obtain resources by swapping around the order of ptr_ and name_, and ensuring that the exception is thrown in the initialiser list. It's reasonable to throw an exception in the initialiser list, but anything managing a resource directly should ideally only manage that one resource.

8.