COMP6771
Advanced C++ Programming

5.1 - Exceptions

In this Lecture

Why?

- Sometimes our programs need to deal with unexpected runtime errors and handle them gracefully.

What?

- Throwing and catching exceptions
- Exception safety levels
- Exception performance

A Motivating Example

- What does the code produce?
- What can go wrong?
  - Call to malloc() in vector could fail?
  - Conversion of string to int could fail when reading in user input?
  - User inputted an index too big?

```
#include <vector>
#include <iostream>

int main() {
  std::cout << “Enter -1 to quit\n”;
  std::vector<int> items = {97, 84, 72, 65};
  std::cout << “Enter an index: ”;
  for (int ix; std::cin >> ix && ix != -1;) {
    std::cout << items[ix] << “\n”;
    std::cout << “Enter an index: ”;
  }
}
```

A (Correct) Motivating Example

- What does the code produce?
- What can go wrong?
  - Call to malloc() in vector could fail
    - unrecoverable error.
  - Conversion of string to int could fail when reading in user input
    - build that into the design of reading from `std::cin`.
  - User inputted an index too big
    - use dynamic exceptions!

```
#include <vector>
#include <iostream>

int main() {
  std::cout << “Enter -1 to quit\n”;
  std::vector<int> items = {97, 84, 72, 65};
  std::cout << “Enter an index: ”;
  for (int ix; std::cin >> ix && ix != -1;) {
    try {
      std::cout << items.at(ix) << “\n”;
    } catch (std::out_of_range &e) {
      std::cout << “Index out of bounds\n”;
    } catch(...) {
      std::cout << “Something went wrong!\n”;
    }
    std::cout << “Enter an index: ”;
  }
}
```

C++ Exceptions

- **Exceptions** are for exceptional circumstances.
  - Problems can happen through execution (things not going to plan A!)
- **Exception Handling**:
  - Run-time (dynamic) mechanism
  - Code downstream detects anomalies and throws an appropriate exception
  - Upstream (sometimes unrelated) code catches the exception, handles it, and potentially rethrows it.
- **Why?**
  - Allows us to gracefully and programmatically deal with anomalies, as opposed to our program crashing.

Conceptual Structure

- Exceptions are just data that is "thrown" up the stack when an error is detected.
  - Usually are instances of classes that extend `std::exception`.
  - Limited type conversions exist for exceptions:
    - From non-const to const (but not the reverse!).
    - User-defined conversions.
- Code that can throw is placed in a `try` block.
- Code to handle a thrown exception is placed in a `catch` block.
- Stack unwinds until an appropriate catch block is found.

```
// for standard exception definitions
#include <exception>

try {
  // Code that can throw
} catch (const std::out_of_range &) {
  // Code that can handle an out-of-range
  // error (maybe from misuse of a vector?)
} catch (const std::exception &) {
  // Code that can catch all exceptions
  // that derive from std::exception
  // (excluding std::out_of_range)
} catch (...) {
  // Code that can catch anything
}
```

Catching the Right Way

- **Throw by value, catch by const reference**
- Ways to catch exceptions:
  - By pointer (no!)
  - By value (no!)
  - By reference (yes !!)
- References are preferred because:
  - more efficient, less copying (exploring today)
  - no slicing problem (related to polymorphism, exploring later)
- Extra reading for those interested

Rethrowing

- When an exception is caught, by default that catch block will be the only part of the code with access to the exception.
- To give other catch blocks upstream access to the exception, can `throw` again.

```
try {
  try {
    std::cout << “oops” << std::endl;
    throw 6771;
  } catch (const int &x) {
    std::cout << “exception detected.”
              << “rethrowing\n”;
    throw x;
  }
} catch (const int &i) {
  std::cout << “i: ” << i << std::endl;
}
//changing exception type
```

Multiple `catch` Blocks

- A single try block can have more than one `catch`.
- Each `catch` block argument is matched against the current exception until an appropriate type is found.
- Should order the catch blocks from most to least specific exception for maximum performance.
- `catch(...)` should almost always be last.

```
#include <iostream>
#include <vector>

int main() {
  auto items = std::vector<int>{};
  try {
    items.resize(items.max_size() + 1);
  } catch (const std::bad_alloc &e) {
    std::cout << "Out of bounds.\n";
  } catch (const std::exception&) {
    std::cout << "General exception.\n";
  } catch (...) {
    std::cout << "Even more general.\n";
  }
}
```

`noexcept`: asking if the exception cant not throw

- Specifies whether a function could potentially throw.
- Querying at compile time-never to emit an exception.
- It doesn't actually prevent a function from throwing an exception.
- If a `noexcept` function throws, `std::terminate` is called.
- Compiler can sometimes make optimisations if it knows a function is `noexcept`.
- Improve algorithmic performance in generic code. `Move-Big(O) faster`
- Precisely provide info, what we need at compile time.

```
class S1 {
public:
  // may throw
  int foo() const;
};

class S2 {
public:
  // does not throw
  int foo() const noexcept;
};

void foo() noexcept {
  // cannot throw either.
}
```

Stack Unwinding

- **Stack Unwinding** is the automatic process of popping stack frames until an appropriate handler for an exception is found.
- Once a found catch block successfully exits without rethrowing, stack unwinding is complete.
- If it catches by value, its formal parameter is initialized by copying the exception object. If it catches by reference, the parameter is initialized to refer to the exception object, then unwinding
- If another exception is thrown during stack unwinding before a `catch` block is found, `std::terminate` is called.

Exceptions & Destructors

- Compiler ensures any destructors are called during stack unwinding.
- All exceptions that occur inside a destructor **must** be handled inside the destructor.
- Therefore, all destructors are implicitly `noexcept` .

Exceptions & Constructors

- What happens if an exception is thrown halfway through a constructor?
- The C++ Standard: "An object that is partially constructed or partially destroyed will have destructors executed for all of its fully constructed subobjects"
- A destructor is not called for an object that was partially constructed.
- ...*except* for an exception thrown in a delegating constructor (why?).
- Common problems:
  - Code that catches an exception thrown in a constructor assumes the object is fully constructed and uses it.
  - A failure partway through an object's construction leads to its destructor running against a partially-constructed object.
  - Resources may be leaked.

```
// spot the bug

class unsafe_class {
public:
   unsafe_class(int a)
   : a_{new int{a}},
   : i_{a == 7 ? a : throw “uh oh”}
   {}

  ~unsafe_class() {
    delete a_;
  }

private:
   int *a_;
   int  i_;
};

int main() {
  auto a = unsafe_class(6771);
}
```

Exception: when/when not ?

- Rare error
- Exceptional case that can not be dealt locally (I/O error)
- File not found
- Cant find key in map
- Operators and constructor where no other mechanism works

- For errors that occurs frequency.
- Function that are expected to fail.

```
auto s_int(std::string const& st)->std::optional<int>
auto s_int(std::string const& st)->boost::outcome<int>
auto s_int(std::string const& st)->std::expected<int>
```

- Have to guarantee response time even in error
- Things that should never happen
  - Use after free
  - Out of range
  - Dereferencing nullptr

std::expected/unexpected

Any given time, it either holds an expected value of type T, or an unexpected value of type E.

Directly allocated within the storage occupied by the expected object. No dynamic memory allocation takes place.

Represents an unexpected value stored in std::expected.

std::expected has constructors with std::unexpected as a single argument, which creates an expected object that contains an unexpected value.

```
std::expected<double, int> ex = std::unexpected(3);
 if (!ex) std::cout << "ex contains an error value\n";
 if (ex == std::unexpected(3)) std::cout << "The error value is equal to 3\n";
```

Exception Safety Levels

- This is not specific to C++.
- It is about writing exception-safe code.
  - Keeping the program in consistent state even after an exception is thrown.
- Operations performed have various levels of safety:
  - No-throw (failure transparency).
  - Strong exception safety (commit-or-rollback).
  - Weak exception safety (no-leak).
  - No exception safety.

No-throw Guarantee

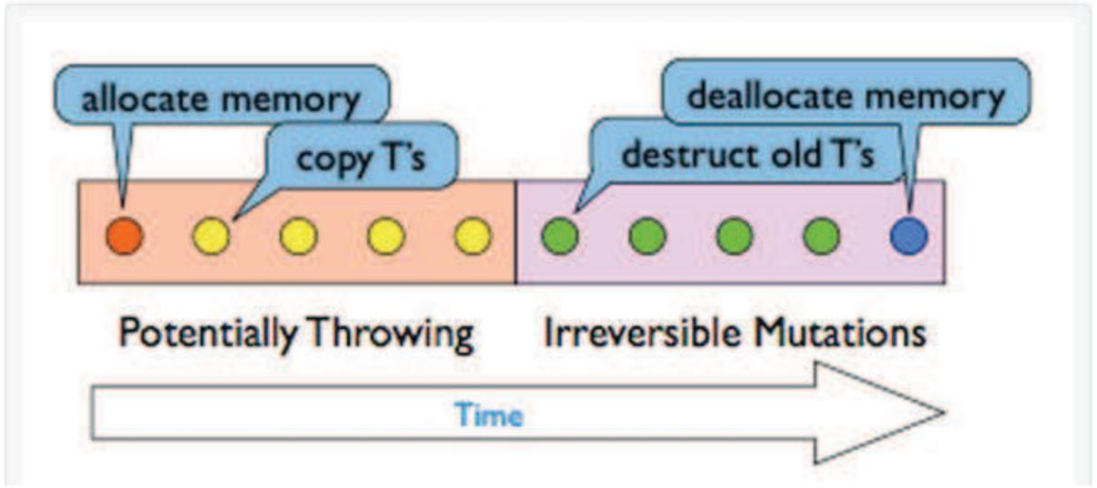
- Also known as failure transparency.
- Operations are guaranteed to succeed, even in exceptional circumstances.
  - Exceptions may occur but are handled internally.
- No exceptions are visible to the client.
- Implemented in C++ with `noexcept`.
- Examples:
  - Closing a file.
  - Freeing memory.
  - Anything done in destructors.
  - Creating a trivial object on the stack (all types from C are trivial).

Strong Exception Safety

- Also known as "commit or rollback" semantics.
- Operations can fail, but failed operations are guaranteed to have no visible effects.
- Probably the most common level of exception safety for types in C++.
- All your copy-constructors should generally follow these semantics.
- Similar for copy-assignment.
  - Can be difficult when manually writing copy-assignment

Implementing Strong Safety

- To achieve strong exception safety, you need to:
  - First perform any operations that may throw, but don't do anything irreversible.
  - Then perform any operations that are irreversible, but don't throw.
  - So-called ”copy & swap” idiom.



```
strong& operator=(strong const& other) {
  strong temp(other); // copy
  temp.swap(*this);   // ...and swap
  return *this;
}
```

Strong guarantee can be costly i,e. if the `Strong` object in the example allocates large amounts of memory. Instead of reusing the already allocated memory, the temporary has to allocate new memory just to release the old one after the swap.

Basic Exception Safety

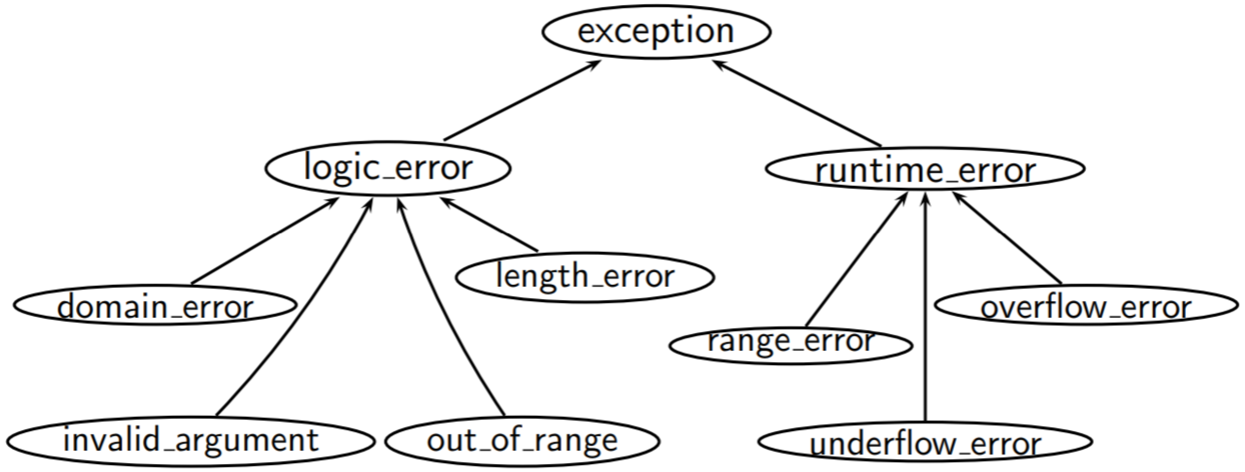
- This is known as the no-leak guarantee.
  - we can be sure that our objects class invariants are not violated. Nothing more, nothing less.
- Change in status of program before exception thrown.
- Partial execution of failed operations can cause side effects, but:
  - All invariants must be preserved.
  - No resources are leaked.
  - No data corruption.
- Any stored data will contain valid values, even if different from before the exception was thrown.
  - A "valid, but unspecified state”.

No Exception Safety

- No guarantees.
- Don't write C++ with no exception safety.
- Very hard to debug when things go wrong.
- Very easy to fix:
  - Be mindful about object lifetimes.
  - Where possible, use the standard library.
  - This gives you basic exception safety for free.

Standard Exceptions

- Standard Library defines its own exception hierarchy.
- If a standard library type throws, it throws one of these exceptions.
- Custom exception objects should derive from the most appropriate standard exception.



Exception Performance

- Exceptions are for exceptional circumstances.
  - They should **not** be used for control flow if a better alternative exists.
  - E.g., instead of throwing an exception, could we return a `std::optional` instead?
- Performance-critical code largely does not use exceptions.
  - A single exception can use 1000 or more CPU cycles before completion.
- Exceptions are best for cold code paths.
- (Optional) Further reading that could improve exception performance.

Feedback