Exception handling

Why exceptions?

- Returning error codes

```
error_code f()
{
  auto rc1 = g1();
  if ( rc1.bad() )
    return rc1;
  auto rc2 = g2();
  if ( rc2.bad() )
    return rc2;
  return g3();
}
```

- Run-time cost
  - small if everything is OK
  - small if something wrong

- Throwing exceptions

```
void f()
{
  g1();
  g2();
  g3();
}
```

- Run-time cost
  - none if everything is OK
  - big if something wrong

Exception handling

- Exceptions are "jumps"
  - Start: ***throw*** statement
  - Destination: ***try-catch*** block
    - Determined in run-time
  - The jump may exit a procedure
    - Local variables will be properly destructed by destructors
  - Besides jumping, a value is passed
    - The type of the value determines the destination
    - Typically, special-purpose classes
    - Catch-block matching can understand inheritance

```
class AnyException { /*...*/ };
class WrongException
 : public AnyException { /*...*/ };
class BadException
 : public AnyException { /*...*/ };
void f()
{
 if ( something == wrong )
   throw WrongException( something);
 if ( anything != good )
   throw BadException( anything);
}
void g()
{
 try {
   f();
 }
 catch ( const AnyException & e1 ) {
   /*...*/
 }
}
```

Exception handling

- Exceptions are "jumps"
  - Start: ***throw*** statement
  - Destination: ***try-catch*** block
    - Determined in run-time
  - The jump may exit a procedure
    - Local variables will be properly destructed by destructors
  - Besides jumping, a value is passed
    - The type of the value determines the destination
    - Typically, special-purpose classes
    - Catch-block matching can understand inheritance
    - The value may be ignored

```
class AnyException { /*...*/ };
class WrongException
  : public AnyException { /*...*/ };
class BadException
  : public AnyException { /*...*/ };
void f()
{
  if ( something == wrong )
    throw WrongException();
  if ( anything != good )
    throw BadException();
}
void g()
{
  try {
    f();
  }
  catch ( const AnyException &) {
    /*...*/
  }
}
```

Exception handling

- Exceptions are "jumps"
  - Start: ***throw*** statement
  - Destination: ***try-catch*** block
    - Determined in run-time
  - The jump may exit a procedure
    - Local variables will be properly destructed by destructors
  - Besides jumping, a value is passed
    - The type of the value determines the destination
    - Typically, special-purpose classes
    - Catch-block matching can understand inheritance
    - The value may be ignored
    - There is an universal catch block

```
class AnyException { /*...*/ };
class WrongException
  : public AnyException { /*...*/ };
class BadException
  : public AnyException { /*...*/ };
void f()
{
  if ( something == wrong )
    throw WrongException();
  if ( anything != good )
    throw BadException();
}
void g()
{
  try {
    f();
  }
  catch (...) {
    /*...*/
  }
}
```

Exception handling

Exception handling

- Evaluating the expression in the throw statement
  - The value is stored "somewhere"
- Stack-unwinding
  - Blocks and functions are being exited
  - Local and temporary variables are destructed by calling destructors (user code!)
  - Stack-unwinding stops in the try-block whose catch-block matches the throw expression type
- catch-block execution
  - The throw value is still stored
    - may be accessed via the catch-block argument (typically, by reference)
  - "throw;" statement, if present, continues stack-unwinding
- Exception handling ends when the accepting catch-block is exited normally
  - Also using return, break, continue, goto
  - Or by invoking another exception

Exception handling

C++11

- Materialized exceptions
  - std::exception_ptr is a smart-pointer to an exception object
    - Uses reference-counting to deallocate
  - std::current_exception()
    - Returns (the pointer to) the exception being currently handled
    - The exception handling may then be ended by exiting the catch-block
  - std::rethrow_exception(p)
    - (Re-)Executes the stored exception
    - like a *throw* statement
  - This mechanism allows:
    - Propagating the exception to a different thread
    - Signalling exceptions in the promise/future mechanism

```
std::exception_ptr p;

void g()
{
 try {
  f();
 }
 catch (...) {
  p = std::current_exception();
 }
}

void h()
{
 std::rethrow_exception( p);
}
```

Exception handling

- Throwing and handling exceptions is slower than normal execution
  - Compilers favor normal execution at the expense of exception-handling complexity
- Use exceptions only for rare events
  - Out-of-memory, network errors, end-of-file, ...

- Mark procedures which cannot throw by ***noexcept***

```
void f() noexcept
{ /*…*/
}
```

  - it may make code calling them easier (for you and for the compiler)
  - ***noexcept*** may be conditional

```
template< typename T>
void g( T & y)
  noexcept( std::is_nothrow_copy_constructible< T>::value)
{
  T x = y;
}
```

Exception handling

- Mark procedures which cannot throw by **noexcept**
- Example: Resizing std::vector<T>
  - When inserting above capacity, the contents must be relocated to a larger memory block
  - Before C++11, the relocation was done by copying, i.e. calling

```
T(const T &)
```

  - If a copy constructor threw, the new copies were discarded and the insert call reported failure by throwing
  - Thus, if the insert threw, no observable change happened
  - Note: Correct destruction of copies is possible only if the destructor is not throwing:

```
~T() noexcept
```

  - In C++11, the relocation shall be done by moving
  - If a move constructor throws, the previously moved elements shall be moved back, but it can throw again!
  - The relocation is done by **moving only if** the move constructor is **declared as**

```
T(T &&) noexcept
```

  - ... **or if** it is declared implicitly and **all elements satisfy** the same property
  - Otherwise, the slower copy method is used!

Exception handling

Standard exceptions

- <stdexcept>
- All standard exceptions are derived from class ***exception***
  - the member function ***what()*** returns the error message
- **bad_alloc**: not-enough memory
- **bad_cast**: dynamic_cast on references
- Derived from logic_error:
  - **domain_error, invalid_argument, length_error, out_of_range**
  - e.g., thrown by vector::at
- Derived from runtime_error:
  - **range_error, overflow_error, underflow_error**

- Hard errors (invalid memory access, division by zero, ...) are NOT signalized as exceptions
  - These errors might occur almost anywhere
  - The need to correctly recover via exception handling would prohibit many code optimizations
  - Nevertheless, there are (proposed) changes in the language specification that will allow reporting hard errors by exceptions at reasonable cost

Exception-safe programming

Exception-safe programming

- Using throw a catch is simple
- Producing code that works correctly in the presence of exceptions is hard
  - Exception-safety
  - Exception-safe programming

```
void f()
{
  int * a = new int[ 100];
  int * b = new int[ 200];
  g( a, b);
  delete[] b;
  delete[] a;
}
```

- If new int[200] throws, the int[100] block becomes inaccessible
- If g() throws, two blocks become inaccessible

Exception-safe programming

```
void f()
{
  int * a = new int[ 100];
  int * b = new int[ 200];
  g( a, b);
  delete[] b;
  delete[] a;
}
```

- If new int[200] throws, the int[100] block becomes inaccessible
- If g() throws, two blocks become inaccessible

- Safety is expensive

```
void f()
{
  int * a = new int[ 100];
  try {
    int * b = new int[ 200];
    try {
       g( a, b);
    } catch (...) {
      delete[] b; throw;
    }
    delete[] b;
  } catch (...) {
    delete[] a; throw;
  }
  delete[] a;
}
```

Exception-safe programming

```
void f()
{
  int * a = new int[ 100];
  int * b = new int[ 200];
  g( a, b);
  delete[] b;
  delete[] a;
}
```

- If new int[200] throws, the int[100] block becomes inaccessible
- If g() throws, two blocks become inaccessible

- Smart pointers can help

```
void f()
{
  auto a = std::make_unique<int[]>(100);
  auto b = std::make_unique<int[]>(200);
  g( &*a, &*b);
}
```

- Exception processing correctly invokes the destructors of smart pointers

Exception-safe programming

- There are more problems besides memory leaks

```
std::mutex my_mutex;

void f()
{
  my_mutex.lock();
  // do something critical here
  my_mutex.unlock();
  // something not critical
}
```

  - If something throws in the critical section, this code will leave the mutex locked forever!

- RAII: Resource Acquisition Is Initialization
  - Constructor grabs resources
  - Destructor releases resources
    - Also in the case of exception

```
std::mutex my_mutex;

void f()
{
  {
    std::lock_guard< std::mutex>
      lock( my_mutex);
    // do something critical here
  }
  // something not critical
}
```

  - There is a local variable “lock” that is never (visibly) used beyond its declaration!
  - Nested blocks matter!

Exception-safe programming

- An **incorrectly** implemented copy assignment

```
T & operator=( const T & b)
{
  if ( this != & b )
  {
    delete body_;
    body_ = new TBody( b.length());
    copy( * body_, * b.body_);
  }
  return * this;
}
```

- Produces invalid object when TBody constructor throws
- Does not work when this==&b

- Exception-safe implementation

```
T & operator=( const T & b)
{
  T tmp(b);
  operator=(std::move(tmp));
  return * this;
}
```

- Can reuse code already implemented in the copy constructor and the move assignment
- Correct also for this==&b
  - although ineffective

Exception-safe programming

- Language-enforced rules
  - Destructors may not end by throwing an exception
  - Constructors of static variables may not end by throwing an exception
  - Move constructors of exception objects may not throw
- Compilers sometimes generate implicit try-catch blocks
  - When constructing a compound object, a constructor of an element may throw
    - Array allocation
    - Class constructors
  - The implicit catch block destructs previously constructed parts and rethrows

Exception-safe programming

Theory

- (Weak) exception safety
  - Exceptions does not cause *inconsistent* state
    - No memory leaks
    - No invalid pointers
    - Application invariants hold
    - ...?
- Strong exception safety
  - Exiting function by throwing means *no change* in (observable) state
  - *Observable state* = public interface behavior
  - Also called "Commit-or-rollback semantics"

Strong exception safety

```
void f()
{
  g1();
  g2();
}
```

- When g2() throws...
  - f() shall signal failure (by throwing)
  - failure shall imply no change in state
  - but g1() already changed something
  - it must be undone

```
void f()
{
  g1();
  try {
    g2();
  } catch(...) {
    undo_g1();
    throw;
  }
}
```

Strong exception safety

- Undoing is sometimes impossible
  - e.g. erase(...)
- Code becomes unreadable
  - Easy to forgot the undo

Observations

- If a function does not change observable state, undo is not required
- The last function in the sequence is never undone

```
void f()
{
  g1();
  try {
    g2();
    try {
      g3();
    } catch(...) {
      undo_g2();
      throw;
    }
  } catch(...) {
    undo_g1();
    throw;
  }
}
```

Strong exception safety

- Check-and-do style
  - Check if everything is correct
  - Then do everything
    - These functions must not throw
  - Still easy to forget a check
  - Work is often duplicated
  - It may be difficult to write non-throwing do-functions

```
void f()
{
    check_g1();
    check_g2();
    check_g3();
    do_g1();
    do_g2();
    do_g3();
}
```

Strong exception safety

- Check-and-do with tokens
  - Each do-function requires a token generated by the check-function
    - Checks can not be omitted
    - Tokens may carry useful data
      - Duplicate work avoided
  - It may be difficult to write non-throwing do-functions

```
void f()
{
    auto t1 = check_g1();
    auto t2 = check_g2();
    auto t3 = check_g3();
    do_g1( t1); // or t1.doit();
    do_g2( t2);
    do_g3( t3);
}
```

Strong exception safety

- Prepare-and-commit style
  - Prepare-functions generate a token
  - Tokens must be committed to produce observable change
    - Commit-functions must not throw
  - If not committed, destruction of tokens invokes undo
  - If some of the commits are forgotten, part of the work will be undone

```
void f()
{
    auto t1 = prepare_g1();
    auto t2 = prepare_g2();
    auto t3 = prepare_g3();
    commit_g1( t1); // or t1.commit();
    commit_g2( t2);
    commit_g3( t3);
}
```

Strong exception safety

- Two implementations:
  - Do-Undo
    - Prepare-functions make observable changes and return undo-plans
    - Commit-functions clear undo-plans
    - Token destructors apply undo-plans
  - Prepare-Commit
    - Prepare-functions return do-plans
    - Commit-functions perform do-plans
    - Token destructors clear do-plans
  - Commits and destructors must not throw
      - Unsuitable for inserting
    - Use Do-Undo when inserting
      - Destructor does erase
    - Use Prepare-Commit when erasing
      - Commit does erase

```
void f()
{
    auto t1 = prepare_g1();
    auto t2 = prepare_g2();
    auto t3 = prepare_g3();
    commit_g1( t1); // or t1.commit();
    commit_g2( t2);
    commit_g3( t3);
}
```

Strong exception safety

- Problems:
  - Some commits may be forgotten
  - Do-Undo style produces temporarily observable changes
    - Unsuitable for parallelism
- Atomic commit required
  - Prepare-functions concatenate do-plans
  - Commit executes all do-plans "atomically"
    - It may be wrapped in a lock_guard
  - Commit may throw!
    - It is the only function with observable effects
- Inside commit
  - Do all inserts
    - If some fails, previous must be undone
  - Do all erases
    - Erases do not throw (usually)

- Chained style

```
void f()
{
   auto t1 = prepare_g1();
   auto t2 = prepare_g2( std::move(t1));
   auto t3 = prepare_g3( std::move(t2));
   t3.commit();
}
```

- Symbolic style

```
void f()
{
   auto t1 = prepare_g1();
   auto t2 = std::move(t1) | prepare_g2();
   auto t3 = std::move(t2) | prepare_g3();
   t3.commit();
}
```