Exception handling

Exception handling

- Exceptions are "jumps"
  - Start: throw statement
  - Destination: try-catch block
    - Determined at 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);
  std::string locvar1;
  if ( anything != good )
    throw BadException( anything);
}

void g()
{
  try {
    std::ofstream locvar2;
    f();
  }
  catch ( const AnyException & e1 ) {
    /*...*/
  }
}
```

Exception handling

- Exceptions are "jumps"
  - Start: throw statement
  - Destination: try-catch block
    - Determined at 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

```cpp
class AnyException { /*...*/ };
class WrongException
  : public AnyException { /*...*/ };
class BadException
  : public AnyException { /*...*/ };

void f()
{
  if ( something == wrong )
    throw WrongException( something);
  std::string locvar1;
  if ( anything != good )
    throw BadException( anything);
}

void g()
{
  try {
    std::ofstream locvar2;
    f();
  }
  catch ( const AnyException &) {
    /*...*/
  }
}
```

Exception handling

- Exceptions are "jumps"
  - Start: throw statement
  - Destination: try-catch block
    - Determined at 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( something);
  std::string locvar1;
  if ( anything != good )
    throw BadException( anything);
}

void g()
{
  try {
    std::ofstream locvar2;
    f();
  }
  catch (...) {
    /*...*/
  }
}
```

Exception handling

- Exception handling consists of
  - 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
      - Inside a destructor, another instance of exception handling may be executed
      - The destructors must not let their internal exceptions escape
    - 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)
      - also accessible through std::current_exception
    - "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 throwing another exception from the catch-block

Exception handling

- Materialized exceptions
  - std::exception_ptr is a smart-pointer to an exception object
    - Uses reference-counting to deallocate
  - std::current_exception()
    - Returns (a pointer to a copy of) 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
    - it may make code calling them easier (for you and for the compiler)
  - You shall always explicitly mark move-constructors and move-assignments as noexcept
    - If you are able to avoid exceptions there
    - It will significantly improve the behavior of containers containing your type
    - Compiler-generated functions will be noexcept if every element has its noexcept function
  - Destructors are noexcept by default
    - If your destructors may throw, you shall mark them noexcept(false)

```
void f() noexcept
{ /*...*/
}
```

- noexcept may be conditional on a compile-time constant
  - Used in conjunction with type-examining *traits* in the standard library

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

Exception handling

- Standard exceptions
  - <stdexcept>
  - All standard exceptions are derived from class std::exception
    - the member function what() returns the error message
  - std::bad_alloc: not-enough memory
  - std::bad_cast: dynamic_cast on references
  - Derived from std::logic_error – usually a mistake of the programmer
    - domain_error, invalid_argument, length_error, out_of_range
    - e.g., thrown by vector::at
  - Derived from std::runtime_error – usually a problem in the data or environment
    - range_error, overflow_error, underflow_error
- It is a good practice to derive your exception classes from std::exception
  - It allows anyone to display the error message by

```
try { /*...*/ } catch (const std::exception & e) { std::cout << e.what(); }
```

- 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
  - Some compilers may be able to do it if asked

Programming with exceptions – basic rules

- Rules of the language
  - Destructors must not end by exception
    - An exception may be invoked inside a destructor, but it must be caught inside
  - Rationale:
    - Stack-unwinding calls destructors of local variables
    - An exception exiting a destructor cannot be reasonably handled
    - If it happens, terminate() is called and the program is exited without finishing the destructors

Programming with exceptions – basic rules

- Rules of the language
  - Destructors must not end by exception
    - An exception may be invoked inside a destructor, but it must be caught inside
  - Technically, this rule applies only to destructors of
    - Local variables (due to exceptions during stack unwinding)
    - Global/static variables (nowhere to catch such exceptions)
  - Logically, it shall be applied anywhere
    - Local variable destructors often call other destructors
    - We don't like objects that refuse to die

Programming with exceptions – basic rules

- Rules of the language
  - Destructors must not end by exception
  - A constructor of global/static variable must not end by exception
    - There is no possibility to catch such exception
    - If it happens, terminate() is called and the program is exited without finishing the destructors
    - Other constructors can safely throw exceptions (and it is a good idea)
    - Avoid global/static variables

Programming with exceptions – basic rules

- Compilers create implicit try-catch blocks
  - Creation of arrays
    - Calls default constructors for every element
    - If the constructor for i-th element throws
      - The (i-1),...,0-th elements are destructed
      - The exception is rethrown - the array is not created
  - Creation of classes/structures
    - Call constructors for every base class and data member
    - If the constructor for an element throws
      - The previous elements are destructed
      - The exception is rethrown - the class is not created
    - The implicit catch block may be augmented with an explicit one:

```
X::X( /* ... */)
try : Y( /* ... */)
{ /* constructor body */
} catch ( /* ... */ ) {
  /* catches all exceptions in both the element constructors and the body
     implicitly rethrows at the end
  */
}
```

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

- The use of smart pointers solves some problems related to exception safety

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

- RAII: Resource Acquisition Is Initialization
  - Constructor allocates resources
  - Destructor frees the resources
    - Even in the case of an exception

```
std::mutex my_mutex;

void f()
{
  std::lock_guard< std::mutex>
  lock( my_mutex);
  // do something critical here
}
```

Programming with exceptions – basic rules

- Catch all exceptions in **main**

```
int main(int argc, char * * argv)
{ try {
    // here is all the program functionality
  } catch (...) {
    std::cout << "Unknown exception caught" << std::endl;
    return -1;
  }
  return 0;
}
```

  - Motivation: "It is implementation-defined whether any stack unwinding is done when an exception is thrown and not caught."
    - If you don't catch in main, your open files may not be flushed, mutexes not released...
  - Insert a std::exception catch block before the universal block to improve diagnostics in known cases

```
  catch (const std::exception & e) {
  { std::cout << "Exception: " << e.what() << std::endl;
    return -1;
  }
```

- This rule does not apply to threads
  - Exceptions in threads launched by **std::thread** are caught by the library
    - These exceptions reappear in another thread if **join** is called
- [Paranoid] A catch with rethrow ensures stack unwinding to this point

```
  try {
    // sensitive code containing write-open files, inter-process locks etc.
  } catch (...) { throw; }
```

Programming with exceptions – basic rules

- Don't consume exceptions of unknown nature
  - You shall always rethrow in universal catch-blocks, except in **main**
  - Also called ***Exception neutrality***

```
void something() {
  try {
    // something
  } catch (...) { // WRONG !!!
    std::cout << "Something happened - but we always continue" << std::endl;
  }
}
```

  - Motivation: It is not a good idea to continue work if you don't know what happened
    - It may mean "hacker attack detected" or "battery exhausted"
- You can consume an exception if you know what parts may be damaged

```
for (;;) {
  auto req = socket.receive_request();
  try {
    auto reply = perform_request( req);
    socket.send_reply(reply);
  } catch (const std::exception & e) { // Any std::exception deemed recoverable
    socket.send_reply(500, e.what());
  }
}
```

  - The damaged parts must be restored or safely disposed of
    - By their destructors during stack-unwinding (preferred)
    - By clean-up code in rethrowing universal catch-blocks (error-prone)

Programming with exceptions – basic rules

- The damaged parts must be restored or safely disposed of
  - By clean-up code in rethrowing universal catch-blocks (error-prone)

```
try {
  some_mutex.lock();
  try {
    auto reply = perform_request( req);
  } catch (...) {
    some_mutex.unlock();
    throw;
  }
  some_mutex.unlock();
  socket.send_reply(reply);
} catch (const std::exception & e) {
  socket.send_reply(500, e.what());
}
```

  - By their destructors during stack-unwinding (preferred)
  - Called *RAII (Resource Acquisition Is Initialization)*

```
try {
  reply_data reply;
  { std::lock_guard g(some_mutex);  // [C++17] template deduction required
    reply = perform_request( req);
  }
  socket.send_reply(reply);
} catch (const std::exception & e) {
  socket.send_reply(500, e.what());
}
```

Programming with exceptions – basic rules

- RAII may require additional exactly positioned blocks in code
- These may interfere with the scope of other declarations

```
try {
  reply_data reply;
  { std::lock_guard g(some_mutex);
    reply = perform_request( req);
  }
  socket.send_reply(reply);
} catch (const std::exception & e) {
  socket.send_reply(500, e.what());
}
```

- May be solved using std::optional

```
try {
  std::optional< std::lock_guard< std::mutex>> g(some_mutex);
  auto reply = perform_request( req);
  g.reset();  // destructs the lock_guard inside
  socket.send_reply(reply);
} catch (const std::exception & e) {
  socket.send_reply(500, e.what());
}
```

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
- Requires testing for 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

- ***(Weak) exception safety***
  - A function (operator, constructor) is *(weakly) safe*, if, after an exception, it leaves all the data in a consistent state
  - Consistent state includes:
    - All unreachable data were properly deallocated
    - All pointers are either null or pointing to valid data
    - All application-level invariants are valid

Exception-safe programming

- ***(Weak) exception safety***
  - A function (operator, constructor) is *(weakly) safe*, if, after an exception, it leaves all the data in a consistent state
  - Consistent state includes:
    - All unreachable data were properly deallocated
    - All pointers are either null or pointing to valid data
    - All application-level invariants are valid

- ***Strong exception safety***
  - A function is *strongly safe*, if, after an exception, it leaves the data in the same (*observable*) state as when invoked
  - *Observable state* - the behavior of the public methods
  - Also called "***Commit-or-rollback semantics***"

Exception-safe programming

- Standard library is designed to be strongly exception-safe, if
  - the user-supplied types/functions are strongly exception-safe
  - some additional conditions hold
  - Example: std::vector::insert
    - *If an exception is thrown when inserting a single element at the end, and T is CopyInsertable or std::is_nothrow_move_constructible<T>::value is true, there are no effects (strong exception guarantee).*
  - The algorithm chosen by the library may depend on **noexcept** flags
    - Insert uses copy-constructors if move-constructors are not marked noexcept
    - Otherwise it would not be able to undo the failed move