

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

- ▶ 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

► 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 and other synchronization features

```
std::exception_ptr p;
```

```
void g()
```

```
{
```

```
  try {
```

```
    f();
```

```
  }
```

```
  catch (...) {
```

```
    p = std::current_exception();
```

```
  }
```

```
}
```

```
void h()
```

```
{
```

```
  std::rethrow_exception( p );
```

```
}
```

▶ Standard exceptions

- `<stdexcept>`
- All standard exceptions are derived from the class `std::exception`
 - the member function **`what()`** returns the error message (as `const char*`)
- **`bad_alloc`**: not-enough memory
- **`bad_cast`**: `dynamic_cast` on references
- **`bad_optional_access`, `bad_variant_access`, `bad_function_call`, ...**
- Derived from `logic_error`:
 - **`domain_error`, `invalid_argument`, `length_error`, `out_of_range`, `future_error`**
 - e.g., thrown by `vector::at`
- Derived from `runtime_error`:
 - **`range_error`, `overflow_error`, `underflow_error`, `regex_error`, `format_error`, ...**
 - **`system_error`** (thrown by stream/filesystem functions when enabled)
- **Hard errors (invalid memory access, division by zero, ...) are NOT signaled as exceptions**
 - These errors might occur almost anywhere
 - The need to correctly recover via exception handling would prohibit many code optimizations
- **Hard errors may also arise in library code - example:**
 - `vector::operator[]` may cause a hard error (or any other behavior) with index out of range
 - `vector::at` will throw `std::out_of_range` with index out of range

- ▶ Language-enforced rules
 - Destructors shall not end by throwing an exception
 - Constructors of static variables shall not end by throwing an exception
 - Move constructors of exception objects shall not throw
 - Violation of these rules cause immediate termination of the program
- ▶ 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

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;
}
```

Programming with exceptions – basic rules

- ▶ Catch all exceptions in **main**
- ▶ 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
 - Similarly, exceptions thrown by functions invoked by **std::async** are propagated to the caller
- ▶ [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 (...) {  
        std::cout << "Something happened - but we always continue" << std::endl;  
        // WRONG !!!  
    }  
}
```

- 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"

Programming with exceptions – basic rules

- ▶ 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());
}
```

Programming with exceptions – basic rules

- The damaged parts must be restored or safely disposed of
 - 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());  
}
```

- The variable ("g") is declared but never explicitly used
- Its purpose is to have a destructor called at the end of the block

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());  
}
```

▶ ***(Weak/basic) exception safety/guarantee***

- 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/guarantee***

- 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***"

▶ ***Nothrow/noexcept guarantee***

- A function that never exits by throwing an exception
 - There may be exceptions thrown and caught inside the function
- Destructors shall have this guarantee
- For other functions, the guarantee may be explicitly marked

T& operator=(T&&) noexcept;

- ▶ Most parts of standard library strives to be strongly exception-safe
 - In templated code, it depends on the properties of the template arguments
- ▶ 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).*
 - Before C++11, relocation for block extension was done by copying
 - 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 non-throwing; however, destructors are non-throwing by default
 - In C++11, the relocation is preferably done by moving
 - If a move constructor throws, the previously moved elements shall be moved back, but it can throw again - the result is an unrecoverable situation!
 - The relocation is done **by moving only** if the move constructor is **declared as `noexcept`**

Exception handling

- 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;
```

```
}
```

▶ Best practices

▶ Default constructor

- Explicit implementation required if there are scalar elements (numbers, pointers)

```
T() noexcept : /*...*/ {}
```

- In most cases, making it **noexcept** is possible
- Prefer the ":" section for explicit initialization (usually to 0/nullptr)
- If all scalar data members are initialized in their declarations, default constructor may be implemented by "= default;"
- It is also safer for other constructors

```
class T { T() = default; int x = /*...*/; U * p = nullptr; /*...*/ };
```

- The explicit default declaration is needed if there are other constructors

▶ Other constructors

- Most non-trivial constructors in non-trivial classes require some allocation
 - Such constructors cannot be **noexcept**
- Constructors that do not allocate (including indirectly through containers) may be marked **noexcept**

```
T( /*...*/ ) noexcept : /*...*/ {}
```

- Don't forget to mark single-parameter constructors **explicit**

▶ Best practices

▶ Destructor

- In a class at the base of an inheritance hierarchy, always create a virtual destructor

```
virtual ~T() {}
```

- Avoid data elements that need clean-up
- If clean-up is really needed, remember the *Rule Of Five*

```
T(T&& b) noexcept : /*...*/ {}
```

```
T& operator=(const T&& b) noexcept { /*...*/ return *this; }
```

```
T(const T& b) : /*...*/ {}
```

```
T& operator=(const T& b) { /*...*/ return *this; }
```

```
~T() { /*...*/ }
```

- Avoid having more than one element that needs clean-up
 - It often requires a try-catch block when working with more than one element that may fail
 - Pack such data elements one-by-one in auxiliary classes
- Destructors are by default non-throwing, the **noexcept** keyword is not used
 - In a destructor, avoid anything that could throw

▶ Best practices

▶ Move constructor, move assignment

- Avoid explicit implementation if possible

```
T(T&&) = default; T& operator=(T&&) = default;
```

- Do not use **noexcept** with **=default**
- It becomes noexcept implicitly if all elements have noexcept move
- Scalar elements (numbers, pointers) implement move by copying, considered noexcept
- Most std library types have noexcept move methods

- If implemented explicitly, always make it **noexcept**

```
T(T&&b) noexcept : /*...*/ { /*...*/ }          T& operator=(T&&b) noexcept { /*...*/ }
```

- Avoid any potentially throwing functionality
- For scalar elements (numbers, pointers), copy and explicitly set the source to 0/nullptr
- For class elements, use the ":" section to invoke move constructors
- The effect on the source object shall be equivalent to invoking the default constructor

▶ Best practices

▶ Copy constructor, copy assignment

- Avoid explicit implementation if possible

```
T(const T&) = default; T& operator=(const T&) = default;
```

- Do not use **noexcept** with **=default**

▶ Exception-safe implementation of copy assignment

```
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
- Can be abbreviated as

```
return operator=(T(b)); // or operator=(auto(b)) since C++23
```



Exception-safe programming

```
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;
    }
}
```

- ▶ 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;
    }
}
```

- ▶ 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();
}
```

- ▶ 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);
}
```

- ▶ 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
 - This is the difference from the check/do style
 - ▶ 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);
}
```

- ▶ 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);
}
```

- ▶ 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();
}
```