Exception Safe Coding

Dirk Hutter
hutter@compeng.uni-frankfurt.de

Prof. Dr. Volker Lindenstruth

FIAS Frankfurt Institute for Advanced Studies

Goethe-Universität Frankfurt am Main,
Germany

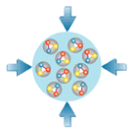
http://compeng.uni-frankfurt.de

C++ User Group

17.06.2015

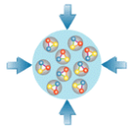
Introduction

- Part 1: Introduction on mechanics of exceptions
- Part 2: Exception safe coding guidelines
- Most material from http://exceptionsafecode.com (Jon Kalb)
- Assume everybody uses at least C++11



Error Management

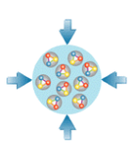
- Logical assertions that by design must be valid …
  - … at compile time: static_assert()
  - … at runtime: assert()
- Error conditions happening at runtime:
  - Return codes
    - Errors are ignored by default
    - If a single call „breaks the chain“ errors are lost
  - Error flagging (errno)
    - Ambiguity which call failed
    - Errors are ignored by default
  - Exceptions
    - Separate error detection from error handling
    - Can’t be ignored
    - Doesn’t mean they are easy to use



Throwing an Exception

```
void f() {
    // this part is executed
    throw std::runtime_error("Error: Something bad happened.");
    // this part is not executed
}
```

- throw e;
  - copy initializes the exception object from e and calls the exception handler
  - e can be any object or a pointer
- All exceptions thrown by the standard library are inherited from std::exception
- Good practice: throw something derived from std::exception by value



Catching an Exception

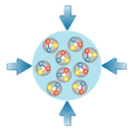
```
catch (A& a) { // catch A by reference
    a.mutating_member();
    throw; // re-throws the exception
}
```

- Parameter to catch works like a function argument
- Exceptions can be modified and re-thrown
- throw;
  - Re-throws the currently handles exception
  - Passes it to the next enclosing exception handler
  - Calls std::terminate if no active exception
- Good practice: catch by reference
  - Otherwise issues with object slicing

Try-Catch-Block

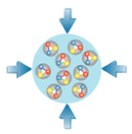
```
try {
    f();
} catch(const std::runtime_error& e) {
    // this executes if f() throws std::runtime_error or something
    // derived form std::runtime_error
} catch(const std::exception& e) {
    // this executes if f() throws something derived from std::exception
} catch(...) {
    // this executes if f() throws any other unrelated type
}
```

- multiple catch blocks can be used
  - are evaluated in order of appearance
  - for re-throw remaining catch blocks will not be evaluated
- catch(…) „catch all“ handler



Stack Unwinding

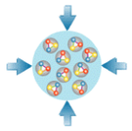
- If an exception is thrown:
- It propagates up the call stack until it reaches a try block
- On its way up destructors for all objects constructed since the try block are called in reverse order of construction
- If a matching catch is found the control flow jumps into this catch block, if not unwinding continues until the next enclosing try block
- If no matching catch can be found or an exception is thrown during stack unwinding std::terminate is called



noexcept

```
void f(); // may throw anything

void g() noexcept(true); // never throws

void g() noexcept; // defaults to ture
```

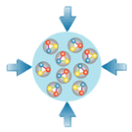
- Specifier whether a function will throw exceptions or not
- If a noexcept function throws std::terminate is called
- Destructors are noexcept by default
- Used for optimization: move if no-throw, else copy
- When should noexcept be used?
  - When it is needed, not for every function
- Note: don’t use dynamic exception specification throw() anymore



noexcept

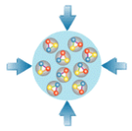
```
inline int f() {return 0;}
static_assert(noexcept( f() ) , ""); // fails!

inline int g() noexcept {return 0;}
static_assert(noexcept( g() ) , ""); // true!
```

- Operator to check at compile time whether a function is declared noexcept



Performance of Exceptions

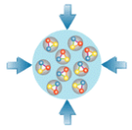
- No throw -> no cost
  - no runtime overhead beside increased code size
- In case of throw
  - Don’t know. Don’t care.
  - Don’t use exceptions to steer your control flow



Function-Try-Block

```
struct S : public R
{
    S() try : R(), foo(2), bar(1) {
        // constuctor body
    }
    catch(...) {
        // catch exceptions form constructor body
        // and initializer list
    } // implicit throw;
}
```

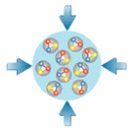
- Associates catch clauses with a function body and the member initializer list (if in constructor)
- Automatically re-throws at the end of the catch clause if in constructor or destructor
- Primarily used to catch and modify exceptions from initializer lists



Function-Try-Block vs. Try-Catch-Block

```
~S()
try // function-try-block
{
    // destructor body
}
catch (...)
{
} // implicit throw;
```

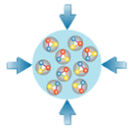
```
~S() {
try // try-catch-block
{
    // destructor body
}
catch (...)
{
}} // nothing special happens here
```



And even more…

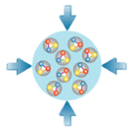
- Nested exceptions
- Passing exceptions between threads
  - std::exception_ptr can be copied between threads

```
int Func(); // might throw
std::future<int> f = std::async(Func());

int v(f.get()); // If Func() threw, it comes out here
```



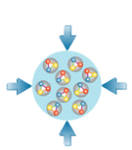
Guidelines

- Throw by value
- Catch by reference
- Use throw; to re-throw
- Use exceptions derived from std::exception



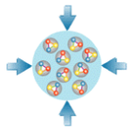
Exception Safety Guarantees

- Formalized by David Abrahams
- **No** exception guarantee
  - If a function throws anything can happen
- **Basic** exception guarantee
  - If an exception is thrown, the object's invariant is still valid and no resource is leaked
- **Strong** exception guarantee
  - If an exception is thrown, the object's state is as it was before the function was called
- **Nothrow** exception guarantee
  - No exception leaves the function



Exception Safe Code

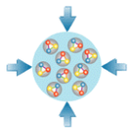
- Everything must support the basic guarantee
- Strong guarantee should be supported where it is natural and comes for free
- Nothrow guarantee in some special cases



Basic guarantee

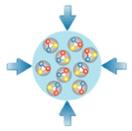
How to not Terminate?

- If terminate is called there is no stack unwinding guaranteed
- std::terminate is called…
- … for unhandled exceptions
- … when re-throw without active exception
- … when noexcept function throws
- … when throwing exception inside active exception
- Destructors might throw!



Guideline

- Destructors must not throw
- Must deliver nothrow exception guarantee
- Cleanup must always be safe
- May throw internally but not emit

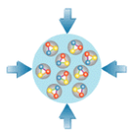


How to ensure not to leak?

- Stack unwinding will destruct all objects on the stack
- We are safe if …

  … every resource is managed by one object on the stack and…

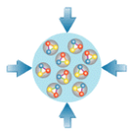
  … every object releases all its resources on destruction
- Use smart pointers to manage heap objects
- Use RAII



Resource Acquisition is Initialisation (RAII)

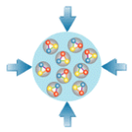
```
class OpenFile {
public:
    OpenFile(const char* filename){
        //throws an exception on failure
        _file.open(filename);
    }

    ~OpenFile(){
        _file.close();
    }

    std::string readLine() {
        return _file.readLine();
    }

private:
    File _file;
};
```

- Resources are acquired in the constructor
  - If you have the object you have the resource
- Resources are released in the destructor
  - Destructors must cleanup all of an objects outstanding responsibilities
- If object is on the stack everything is cleaned up properly



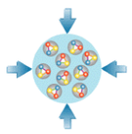
Guidelines

- Use RAII
- Every object should manages only one resource
- Be sure all cleanup code is called by a destructor
- Cleanup must not throw
- Objects may have a additional release function



Aborted Construction

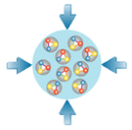
- What happens if an exception is thrown while constructing an object?
  - Base classes and data members are cleaned up by the runtime
  - Everything we do in the constructor body needs to be cleaned up by us
- Guideline: Assign ownership of every resource, immediately upon allocation, to a named manager object that manages no other resources.



Assigning Resources to Smart Pointers

```
FooBar(smart_ptr<Foo>(new Foo(f)),
       smart_ptr<Bar>(new Bar(b)));
```

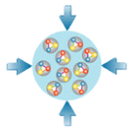
- Both objects are assigned to a smart pointer
- Is this safe?



Assigning Resources to Smart Pointers

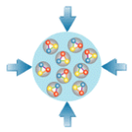
```
auto r(std::make_shared<Foo>(f));
auto s(std::make_unique<Foo>(f)); // C++14

template<typename T, typename ...Args>
std::unique_ptr<T> make_unique( Args&& ...args )
{
        return std::unique_ptr<T>( new T( std::forward<Args>(args)... ) );
}
```

- For smart pointers use make_shared, make_unique
  - More efficient
  - Safer



Leaking Object Memory

- If new calls an constructor that throws the matching operator delete is called
  - Object memory is not leaked
- For placement new with user defined args (other than void*) there is no matching delete operator by default
  - Memory is leaked
- Define matching „placement“ delete operator



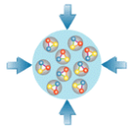
Strong Exception Guarantee

```
void FunctionWithStrongGuarantee() {
// Code That Can Fail
ObjectsThatNeedToBeModified.MakeCopies(OriginalObjects);
ObjectsThatNeedToBeModified.Modify();
```

critical line

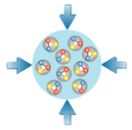
```
// Code That Cannot Fail (Has a No-Throw Guarantee) !
ObjectsThatNeedToBeModified.swap(OriginalObjects); }
```

- Think about a critical line
  - Everything above can throw but doesn't modify the original data
  - Everything below must not throw
- Key function: swap with nothrow guarantee
  - available if move of an object in nothrow



Where to catch?

- Switch
  - Anywhere you need to switch the method of error reporting
  - Where notrhrow needs to be supported, C-APIs, other exception types
- Strategy
  - Anywhere you have a strategy to deal with an error, e.g. a fallback method
- Some Success
  - Anywhere that partial failure is acceptable

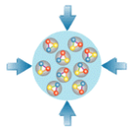


Exception Safety Guidelines

- Throw by value, catch by reference
- Put try-catch-block with catch all in main
- Destructors must not throw
- User RAII
  - Every responsibility is an object
  - All cleanup code must be called by a destructor
- Use critical line for strong exception guarantee
- Know where to catch

Resources

- http://exceptionsafecode.com
- http://en.cppreference.com
- https://www.wikipedia.org
- http://herbsutter.com/gotw/_102/
- http://www.tomdalling.com/blog/software-design/resource-acquisition-is-initialisation-raii-explained/



Thanks for your attention

Dirk Hutter
hutter@compeng.uni-frankfurt.de

CBM