Using Exceptions in C++

a practical guide

18 April 2018

Zbigniew Skowron

Agenda

1. Introduction
2. Exception Basics
3. Motivation
4. Exception Safety
5. Technicalities
6. What to throw and when to catch
7. Exception propagation
8. Adding information to exceptions
9. Thread interruption
10. More technicalities

Introduction

What is the most important feature of C++?

One that enables us to write correct code:

- Destructors

They enable us to enforce invariants:

- Automatically
- Deterministically
  - at well specified time
  - in well specified order

C, C#, Java, Python, Ruby, Scala, Go, TypeScript etc. – they don't have this!
(Rust has.)

RAII - Resource Acquisition Is Initialization

- create "handle" objects for all resources,
- those objects will release resources in destructors.

```
void main()
{
    FILE* f = fopen("ala.txt");

    ...

    fclose(f)
}
```

Evil

Leaks in case of:

- return,
- throw,
- break,
- continue,
- goto...

```
void main()
{
    fstream file("ala.txt");

    ...

}
```

100% safe!

Examples:

- unique_ptr (memory),
- fstream (file handle),
- mutex (critical section),
- thread (operating system thread)
- etc.

C++ Core Guidelines E.6

Use RAII religiously, everywhere*

RAII is the only* way to write correct code.

It’s also critical for writing exception-safe code.

Why some game programmers hate RAII?

Destructors are performing operations on one object at a time.
If you have many objects this is slow.

- Use resource pools.
- Do cleanup once per frame.
- Use data oriented design.

CppCon 2014: Mike Acton "Data-Oriented Design and C++"

Exception Basics

What are exceptions?

Exceptions are like a return:

```
return 30;
throw 30;
```

They both are meant to inform about the result of the function call.

Return reports value of a successful invocation.

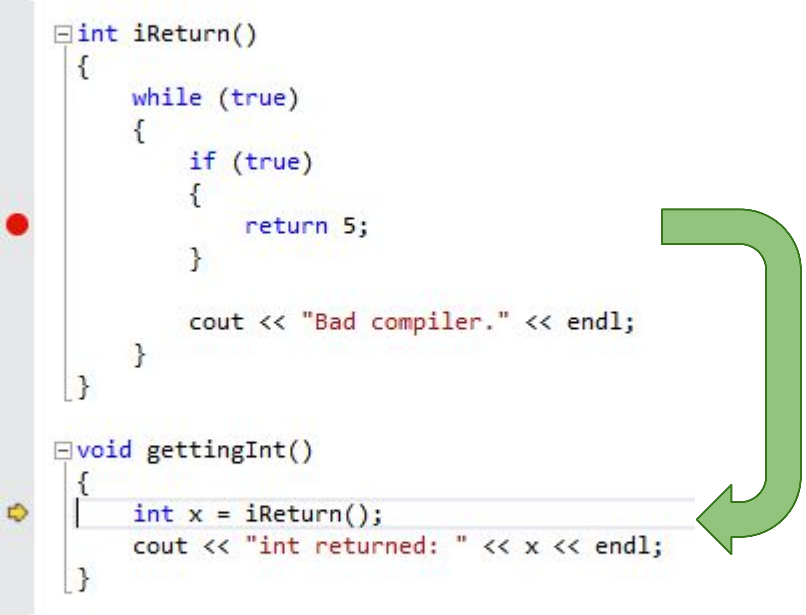
Exceptions report failures during invocation.

They are:

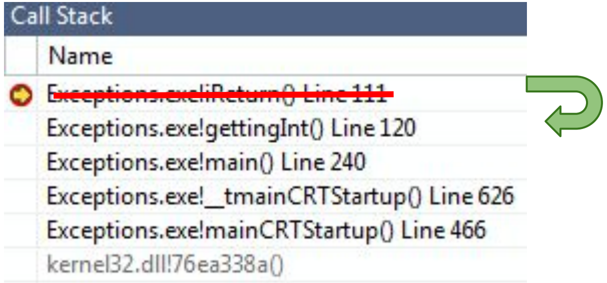
- more expressive than traditional ints,
- impossible to ignore,
- systematic.

Return

Return exits to the calling point.



```
int iReturn()
{
    while (true)
    {
        if (true)
        {
            return 5;
        }

        cout << "Bad compiler." << endl;
    }
}

void gettingInt()
{
    int x = iReturn();
    cout << "int returned: " << x << endl;
}
```

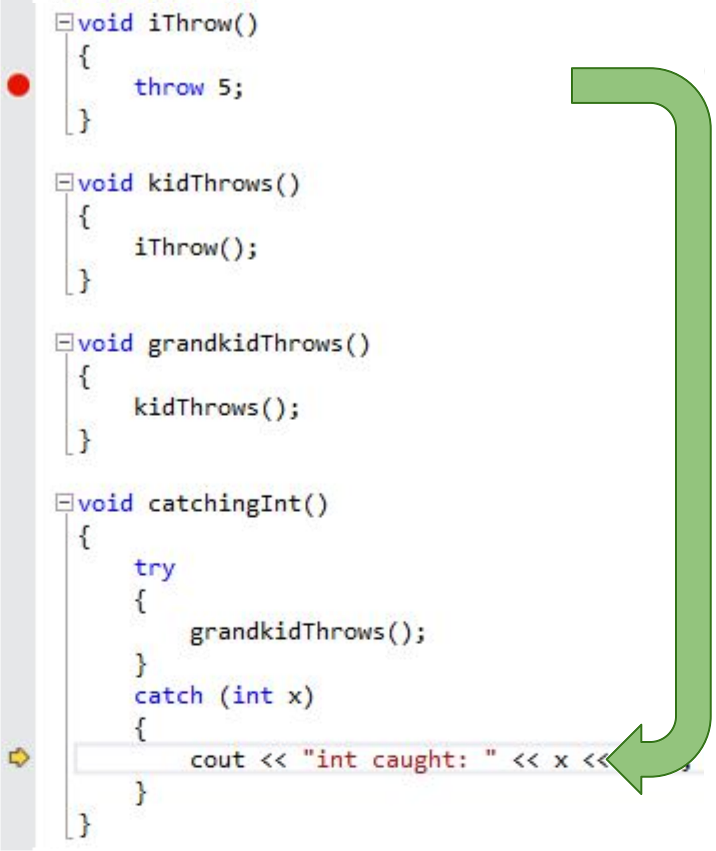


Call Stack

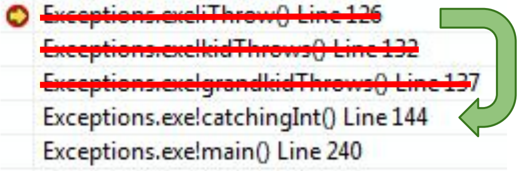
Name
~~Exceptions.exe!iReturn() Line 111~~
Exceptions.exe!gettingInt() Line 120
Exceptions.exe!main() Line 240
Exceptions.exe!__tmainCRTStartup() Line 626
Exceptions.exe!mainCRTStartup() Line 466
kernel32.dll!76ea338a()

Throw

Throw exits to nearest enclosing catch.



```
void iThrow()
{
    throw 5;
}

void kidThrows()
{
    iThrow();
}

void grandkidThrows()
{
    kidThrows();
}

void catchingInt()
{
    try
    {
        grandkidThrows();
    }
    catch (int x)
    {
        cout << "int caught: " << x <<
    }
}
```



~~Exceptions.exe!iThrow() Line 126~~

~~Exceptions.exe!kidThrows() Line 132~~

~~Exceptions.exe!grandkidThrows() Line 137~~

Exceptions.exe!catchingInt() Line 144

Exceptions.exe!main() Line 240

Nearest catch

C++ Core Guidelines E.31

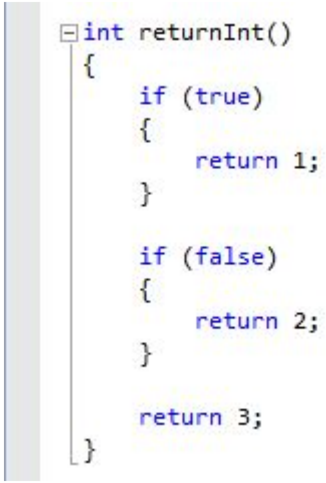
Exception is caught be the nearest matching enclosing catch.
So order of catches is important.

```cpp
void catching()
{
    try
    {
        throw std::runtime_error("Ooops.");
    }
    catch (...)
    {
        cout << "Something else.";
    }
    catch (const std::exception&)
    {
        cout << "std::exception";
    }
    catch (const std::runtime_error&)
    {
        cout << "std::runtime_error";
    }
}
```

```cpp
void catching()
{
    try
    {
        throw std::runtime_error("Ooops.");
    }
    catch (const std::exception&)
    {
        cout << "std::exception";
    }
    catch (const std::runtime_error&)
    {
        cout << "std::runtime_error";
    }
}
```

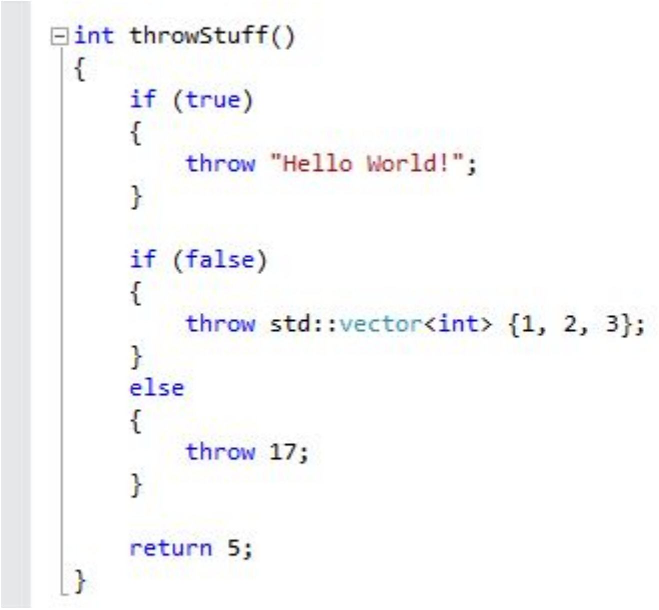
Exceptions are very flexible

- Return has a fixed type of argument.



```
int returnInt()
{
    if (true)
    {
        return 1;
    }

    if (false)
    {
        return 2;
    }

    return 3;
}
```

- Throw can throw anything.

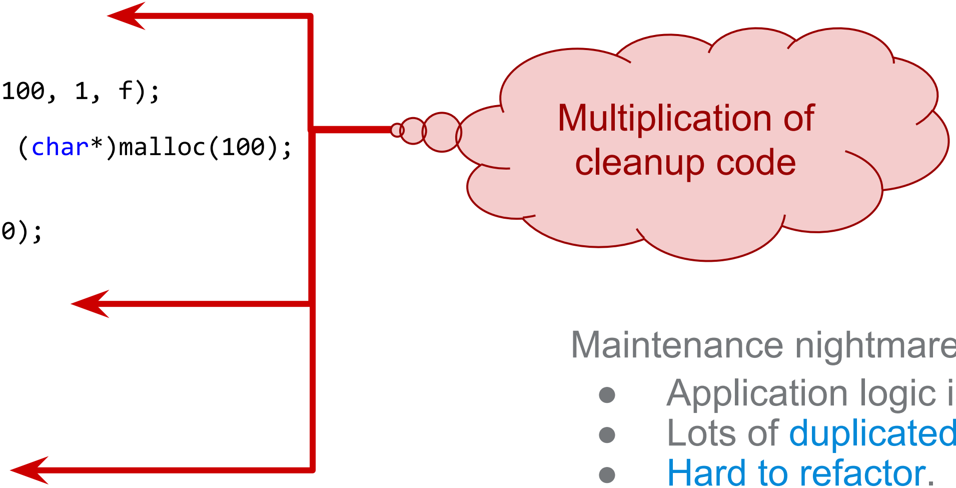


```
int throwStuff()
{
    if (true)
    {
        throw "Hello World!";
    }

    if (false)
    {
        throw std::vector<int> {1, 2, 3};
    }
    else
    {
        throw 17;
    }

    return 5;
}
```

Motivation

C code example

```c
int doStuff0()
{
    char* buffer0 = (char*)malloc(100);
    if (!buffer0)
        return -1;

    FILE* f = fopen("file.txt", "rb");
    if (!f)
    {
        free(buffer0);
        return -2;
    }

    fread(buffer0, 100, 1, f);

    char* buffer1 = (char*)malloc(100);
    if (!buffer1)
    {
        free(buffer0);
        fclose(f);
        return -1;
    }

    free(buffer0);
    free(buffer1);
    fclose(f);

    return 0;
}
```



Multiplication of cleanup code

Maintenance nightmare:

- Application logic is completely obscured.
- Lots of duplicated code.
- Hard to refactor.
- Easy to ignore errors.
- Many places where a mistake can happen.

C code example - “improved”

```
int doStuff1()
{
    int err = 0;

    char* buffer0 = (char*)malloc(100);
    if (!buffer0)
        return -1;

    FILE* f = fopen("file.txt", "rb");
    if (!f)
    {
        err = -2;
        goto free1;
    }
    fread(buffer0, 100, 1, f);

    char* buffer1 = (char*)malloc(100);
    if (!buffer1)
    {
        err = -2;
        goto free2;
    }

    free(buffer1);
free2:
    fclose(f);
free1:
    free(buffer0);

    return err;
}
```

Cleanup code is not duplicated any more!

Yay!

Maintenance nightmare:

- Application logic is still obscured.
- Goto’s are just too easy to break - there is no structure that the compiler can check.
- Hard to refactor.
- Easy to ignore errors.
- Many places where a mistake can happen.

C++ - using RAII for cleanup

```
int doStuff1()
{
    int err = 0;

    char* buffer0 = (char*)malloc(100);
    if (!buffer0)
        return -1;

    FILE* f = fopen("file.txt", "rb");
    if (!f)
    {
        err = -2;
        goto free1;
    }
    fread(buffer0, 100, 1, f);

    char* buffer1 = (char*)malloc(100);
    if (!buffer1)
    {
        err = -2;
        goto free2;
    }

    free(buffer1);
free2:
    fclose(f);
free1:
    free(buffer0);

    return err;
}
```

```
int doStuff2()
{
    unique_ptr<char[]> buffer0(new char[100]);

    fstream f("file.txt");
    if (!f.is open())
        return -1;

    f.read(buffer0.get(), 100);
    if (!f.good())
        return -2;

    unique_ptr<char[]> buffer1(new char[100]);

    return 0;
}
```

C++ - using exceptions for error handling

```
int doStuff1()
{
    int err = 0;

    char* buffer0 = (char*)malloc(100);
    if (!buffer0)
        return -1;

    FILE* f = fopen("file.txt", "rb");
    if (!f)
    {
        err = -2;
        goto free1;
    }
    fread(buffer0, 100, 1, f);

    char* buffer1 = (char*)malloc(100);
    if (!buffer1)
    {
        err = -2;
        goto free2;
    }

    free(buffer1);
free2:
    fclose(f);
free1:
    free(buffer0);

    return err;
}
```

```
void doStuff3()
{
    unique_ptr<char[]> buffer0(new char[100]);

    fstream f("file.txt");
    f.exceptions(std::ifstream::failbit);
    f.read(buffer0.get(), 100);

    unique_ptr<char[]> buffer1(new char[100]);
}
```

- Only code that actually does the job.
- Short.
- Easy to understand.
- Easy to refactor.
- All errors are handled.
- All resources are freed.

C example - error handling

```c
char* readFile(const char* fileName)
{
    char* buffer = (char*)malloc(100);
    if (!buffer)
        return NULL;

    FILE* f = fopen(fileName, "rb");
    if (!f)
    {
        free(buffer);
        return NULL;
    }

    fread(buffer, 100, 1, f);
    fclose(f);

    return buffer;
}
```

Error codes contain very little information:

- Very little information about what happened:
  - Often one error code is used for many different causes (EFAIL, EINVAL, EPERM, Unknown Error).
- No information about context:
  - Which file was not found?
  - Why did we even try to open it?
  - What permissions to which resource were denied, any why did we even try to get it?
- Often ignored.

Rich information in exceptions

```
char* readFile(const char* fileName)
{
    smart ptr<char> buffer = allocate(100);
    if (!buffer)
        throw OutOfMemory("Tried to allocate buffer while reading file: ", fileName);

    smart file f(fileName);
    if (!f)
        throw FileNotFound("Could not open file: ", fileName);

    f.read(buffer);

    return buffer.release();
}
```

Now we know exactly what happened and why.

It’s now much easier to analyze problems.

Better example

```
smart_array<char> readFile(const char* fileName)
{
    auto buffer = allocate(100);

    smart file f(fileName);
    f.read(buffer);

    return buffer;
}
```

All error handling and cleanup is encapsulated.

In contrast: in C most of the code is error handling and cleanup.

```
char* readFile(const char* fileName)
{
    char* buffer = (char*)malloc(100);
    if (!buffer)
        return NULL;

    FILE* f = fopen(fileName, "rb");
    if (!f)
    {
        free(buffer);
        return NULL;
    }

    if (fread(buffer, 100, 1, f) != 100)
    {
        free(buffer);
        fclose(f);
        return NULL;
    }

    fclose(f);

    return buffer;
}
```

Conclusions from examples

In C:

- Cleanup code mixed with and obscuring program logic.
- Error handling mixed with and obscuring program logic.
- Code is mostly error handling and cleanup.
- Limited expressiveness of error handling:
    - you need to fit function return value and error code into one value (bad),
    - or use out parameters (ugly),
    - or use static storage like errno (evil).
- Easy to ignore errors.
- Error prone.

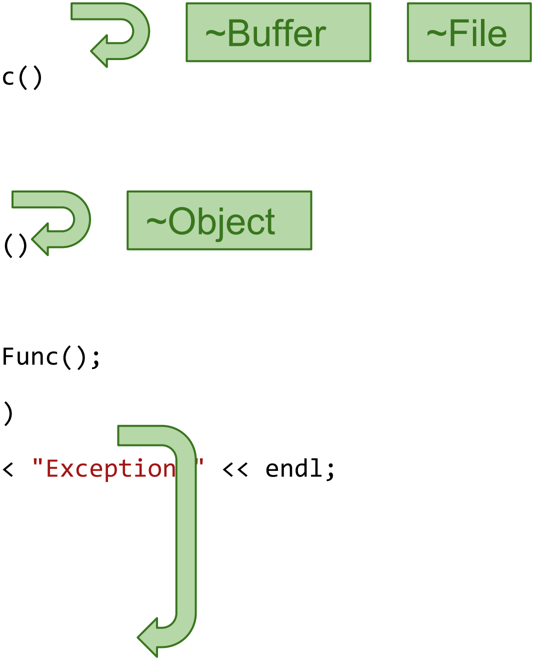
In C++:

- Visible program logic.
- Automatic cleanup using destructors.
- Transparent error handling using exceptions.
- Very expressive error handling.
- All errors are handled.
- Bug free.*

Cleanup code

Stack unwinding - return

```
void badFunc()
{
    File f;
    Buffer b;

    return;
}

void victimFunc()
{
    Object o;
    badFunc();
}

void catchFunc()
{
    try
    {
        victimFunc();
    }
    catch (...)
    {
        cout << "Exception" << endl;
    }
}
```



~Buffer

~File

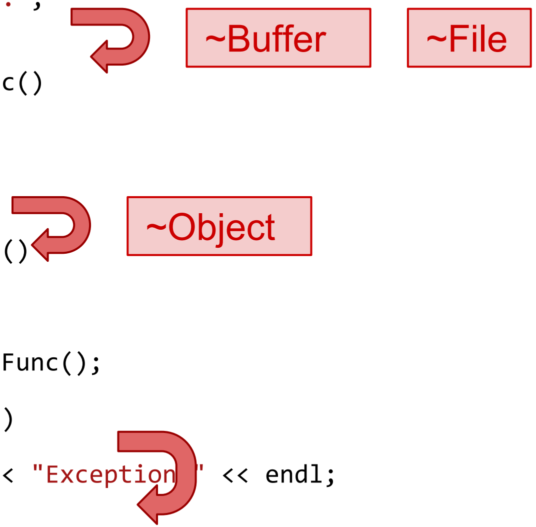
~Object

When a function returns, the stack is "unwinded", which means, that all stack frames are destroyed, one by one.

All local objects are destroyed in order.

Stack unwinding - exception

```
void badFunc()
{
    File f;
    Buffer b;

    throw "Bad.";
}

void victimFunc()
{
    Object o;
    badFunc();
}

void catchFunc()
{
    try
    {
        victimFunc();
    }
    catch (...)
    {
        cout << "Exception" << endl;
    }
}
```



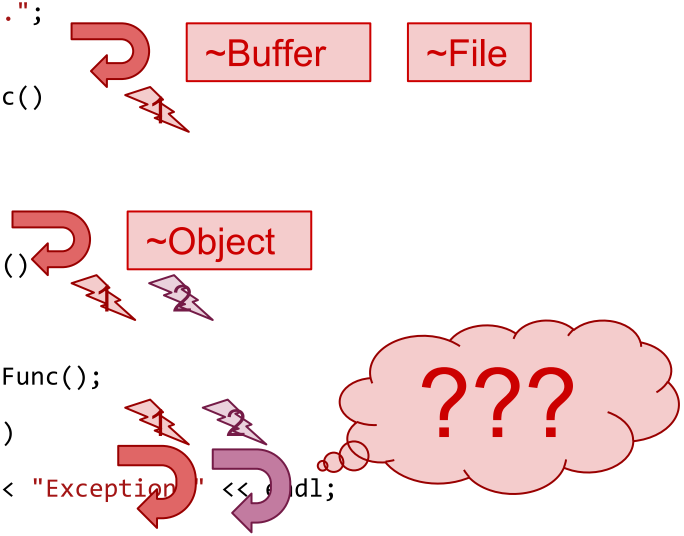
When an exception is thrown, the stack is "unwinded", which means, that all stack frames are destroyed, one by one.

All local objects are destroyed as the exception is leaving their scope.

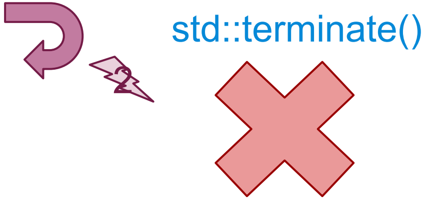
This exactly the same mechanism, as when return is used. No magic here!

Throwing from a destructor

```
void badFunc()
{
    File f;
    Buffer b;

    throw "Bad.";
}

void victimFunc()
{
    Object o;
    badFunc();
}

void catchFunc()
{
    try
    {
        victimFunc();
    }
    catch (...)
    {
        cout << "Exception" << endl;
    }
}
```



```
struct Object
{
    ~Object()
    {
        throw "Bad Object!";
    }
};
```



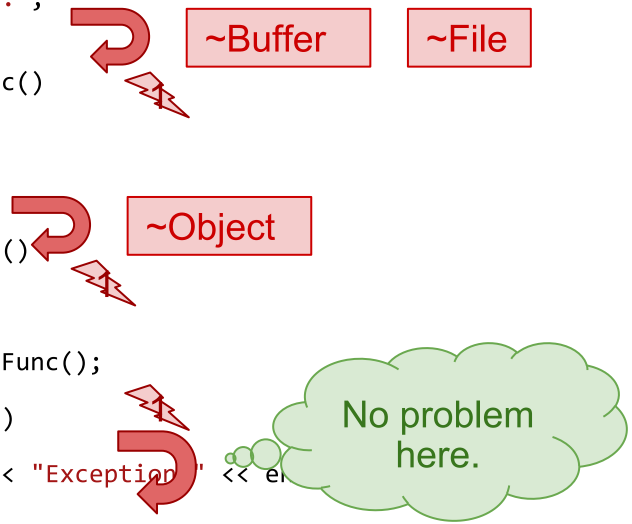
Since having two exceptions in-flight at the same time would we weird, it is explicitly forbidden by the standard.

If you will do this, std::terminate() will be called, and your app will die.

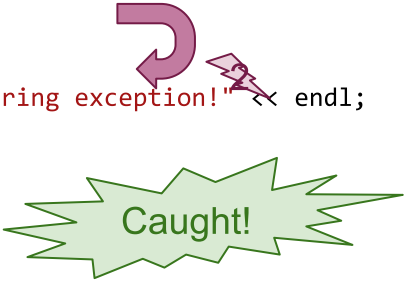
So don't throw from destructors.

Throwing inside a destructor

```
void badFunc()
{
    File f;
    Buffer b;

    throw "Bad.";
}

void victimFunc()
{
    Object o;
    badFunc();
}

void catchFunc()
{
    try
    {
        victimFunc();
    }
    catch (...)
    {
        cout << "Exception" << e
    }
}
```



```
struct Object
{
    ~Object()
    {
        try
        {
            throw "Bad Object!";
        }
        catch (...)
        {
            cout << "Ignoring exception!" << endl;
        }
    }
};
```



You can throw inside a destructor, as long as the exception will not escape from it.

Where not to throw

C++ Core Guidelines

E.16: Destructors, deallocation, and swap must never fail

- C.36: A destructor may not fail
- C.66: Make move operations noexcept
- C.84: A swap function may not fail
- C.85: Make swap noexcept

Think about it:

if cleanup code can fails, then the only thing you can do is kill the program anyway.

“We don't know how to write reliable programs if a destructor, a swap, or a memory deallocation fails.”

The standard library assumes that destructors, deallocation functions (e.g., operator delete), and swap do not throw. If they do, basic standard-library invariants are broken.

C.89: Make a hash noexcept

C.86: Make == symmetric with respect to operand types and noexcept

Destructor design guideline

If anything in your destructor might throw, you have to catch all exceptions.

This means, that errors will be ignored.
Isn't that a bad thing?

Maybe. But there is not much we can do about it apart from:

- Designing your cleanup code to never fail.
- Making sure that errors that are important will be thrown earlier.

```
struct Object
{
    ~Object()
    {
        someStuffThatThrows();
    }
};
```

❌

```
struct Object
{
    ~Object()
    {
        try
        {
            someStuffThatThrows();
        }
        catch (...)
        {
            cout << "Ignoring exception!" << endl;
        }
    }
};
```

Designing your cleanup code to never fail

Designing your cleanup code to never fail.
Easy:

- fclose() - guaranteed not to fail.
- free() - guaranteed not to fail.
- delete p - guaranteed not to fail.
- etc…

Hard:

- RPC

Cleanup earlier, or ignore errors

C++ Core Guidelines E.16

If you want to provide cleanup code that can fail:

- Add a close() method that can throw.
- Call it in destructor, but ignore errors.

This way if the user is interested in cleanup errors, he can handle them explicitly.

Otherwise they are ignored.

```
struct Object
{
    void close()
    {
        someStuffThatThrows();
    }

    ~Object() noexcept
    {
        try
        {
            close();
        }
        catch (...)
        {
            cout << "Ignoring exception!" << endl;
        }
    }
};
```

Cleanup code - scope guard

C++ Core Guidelines E.19

Sometimes it can be useful to have an ad-hoc cleanup code, for example when using C libraries.

Scope guards can be used for this.

```
void usingC()
{
    void* obj = gst_alloc_obj();
    SCOPE_EXIT(gst_free_obj(obj));

    gst_open(obj);
    SCOPE_EXIT(gst_close(obj));

    doStuff(obj);
    maybeThrow(obj);
    doMoreStuff(obj);
}
```

~gst_close(obj)

~gst_free_obj(obj)

Scope guard is a destructor

But remember: instructions in scoped guard are executed in it’s destructor.

So they must not throw!

```
void usingC()
{
    ...
    BOOST_SCOPE_EXIT()
    {
        try
        {
            someStuffThatThrows();
        }
        catch (...)
        {
            cout << "Ignoring exception!" << endl;
        }
    };
    ...
}
```

Constructor design guideline

C++ Core Guidelines E.5

If an error will happen in constructor - throw.
But be careful...

```
struct Object
{
    int* ptr;

    Object()
    {
        ptr = new int[100];

        loadData(ptr);
    }

    ~Object()
    {
        delete ptr;
    }
};
```

Memory leak if loadData() throws.

Destructor is called only if constructor will succeed.

```
struct Object
{
    unique_ptr<int[]> ptr;

    Object()
        : ptr(new int[100])
    {
        loadData(ptr);
    }
};
```

Use RAII. Always.

Use one RAII object for each resource.
Don’t bundle them together, or you will face leaks.

Guidelines

If an error will happen in constructor - throw.
Don't leave an object in a bad state, and provide bool isValid() method.

All your objects should either be in good state, or should not be created at all.

```
auto p = new Object();
```

Don't worry. If constructor throws the memory is freed by new operator.

Avoid writing functions that return bool.
Most often bool is used as an error code:
bool parseFile()

Use exceptions for reporting errors.

Exception Safety

Exception safety guarantees

- No throw guarantee
  Operation will not throw.

- Strong exception safety
  Operation will either succeed, or be rolled back.

Like a database transaction.

- Basic exception safety
  No resource leaks, invariants preserved,
  but operation can be half done.

Be careful!

- No exception safety
  If you will throw exceptions bad things will happen.

Segfaults, memory corruption, painful death...

Exception safety examples

- No throw guarantee
  C functions, like fclose(), can’t throw.
  std::swap(a, b) (*terms and conditions apply)

- Strong exception safety
  std::vector::push_back(a)

Either element will be added, or not.

- Basic exception safety

```
void halfDoneCopy(const std::vector<A>& in, std::vector<A>& out)
{
    for (const auto& a : in)
        out.push_back(a);
}
```

If copy constructor will throw, only some elements will be copied.

- No exception safety

```
char* noSafety(const A& a)
{
    char* buffer = new char[100];
    a->fill(buffer);
    return buffer;
}
```

If fill() will throw we have a memory leak.

Basic exception safety

No resource leaks, invariants preserved, but operation can be half done. Let’s look at this code:

```
struct Map
{
    vector<int> keys;
    vector<string> values;

    void insert(int key, string value)
    {
        keys.push_back(key);
        values.push_back(value);
    }

    string get(int key)
    {
        auto pos = std::find(keys.begin(), keys.end(
        if (pos == keys.end()) return {};
        return values[ pos - keys.begin() ];
    }
};
```

Does insert() guarantee basic exception safety?

insert() should be documented to state invariant:
sizes of keys and values should be the same.

bad_alloc here violates the invariant!

After that it’s not safe to use the object.

Map is not exception safe!

It’s not enough to use RAII, to be exception safe.
Make extra care to uphold any invariants that you have defined.

Strong exception safety

Basic idea of writing exception safe code is as follows:
Separate the function into two parts:

- Prepare: can throw, but does not modify the object.
- Commit: doesn't throw, and modifies the object.

A very inefficient, but illustrative way of making previous code strongly exception safe:

```
void insert(int key, string value)
{
    auto newKeys = keys;
    auto newValues = values;
    newKeys.push_back(key);
    newValues.push_back(value);

    std::swap(newKeys, keys);
    std::swap(newValues, values);
}
```

Prepare.
Can throw, but object is not modified

Commit.
Doesn't throw.

Strong exception safety

Other ways of achieving strong exception safety:

```
void insert(int key, string value)
{
    keys.push_back(key);
    try
    {
        values.push_back(value);
    }
    catch (...)
    {
        keys.pop_back(key);
    }
}
```

This works, because vector guarantees strong exception safety for push_back() and no-throw for pop_back().

```
void insert(int key, string value)
{
    keys.push_back(key);
    ON_FAILURE(keys.pop_back(key));
    values.push_back(value);
}
```

But in general it’s very difficult to write strongly exception safe code.
If only you can separate function into prepare and commit phases - do it.
Otherwise be very careful.

Writing exception safe code

- No throw guarantee
  Don’t throw, and don’t use anything, that throws.
  Use noexcept to mark functions as non-throwing.

- Strong exception safety
  Very difficult.
  Separate your functions into prepare and commit phases if possible.
  std::swap() can help with the commit code.

- Basic exception safety
  Use RAII.
  Make sure your destructors don’t throw.
  Make extra care to uphold any invariants that you have defined.

- No exception safety
  Use RAII everywhere, even if you don’t use exceptions.

Why people don’t use exceptions

Why people don’t use exceptions

- I’ve never used them and I don’t know how.
- I don’t know what will happen.
  - We know exactly what will happen. It’s exactly like return.
- Number of possible paths through code increases
  - It doesn’t, unless you ignore errors.
- Number of possible program states increases.
  - Number of possible program states is largely irrelevant.
  - As long as the invariants hold, it’s fine.
  - That is a big problem in Java, C#, Python etc. where we can’t enforce invariants by program construction.
- I don’t see where my code may be interrupted, so I can’t write correct code.
- If everything can throw, I don’t know how to write correct code.
  - You need to pay attention to what throws only in places, where you violate invariants.
  - In all the rest of the code - you don’t care.
- It’s slow.
  - Turning on exceptions makes C programs slower by ~3% (maybe).
  - Rewriting code to use exceptions can recuperate those losses. Or not. There are no trustworthy benchmarks for that.
  - Throwing exceptions is slow. Fact.

Good reasons not to use exceptions

C++ Core Guidelines E.6

There are many anti-exceptions myths around.
We know of only a few good reasons:

- You have 2K of memory.
- You are in hard-real-time.
- You have spaghetti code.
- Your C++ compiler sucks.
- You'll get fired for challenging your manager's ancient wisdom.

Technicalities

How exceptions are implemented

There are two main approaches:

1. Dynamic construction of list of all cleanup actions that need to be called.

2. Static tables, generated during compilation.
   - compressed by generating them as VM code

Itanium ABI - two phase unwinding:

- search and terminate if catch was not found,
- unwind and cleanup.

Videos:
C++ Exception Handling - The gory details of an implementation
CppCon 2017: Dave Watson “C++ Exceptions and Stack Unwinding”

Godbolt

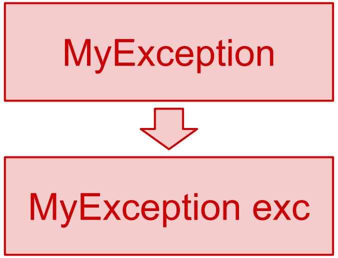
Catch by const reference

Exceptions should be caught by const reference.
Consider this example:

```
void catchFunc()
{
    try
    {
        throw MyException();
    }
    catch (MyException exc)
    {
        cout << "Exception!" << endl;
    }
}
```

MyException is stored on the side

MyException is copied



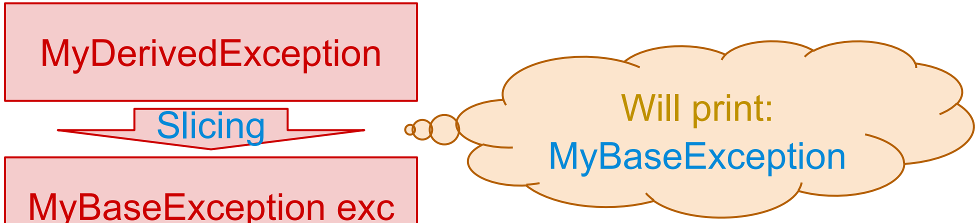
Copy is unnecessary here. So just catch by reference.
Use const to underline the fact, that you don’t modify the exception.

Catch by const reference - slicing

C++ Core Guidelines E.15

Slicing can happen when catching by value the same way it happens when passing parameters by value. Consider this example:

```
void catchFunc()
{
    try
    {
        throw MyDerivedException();
    }
    catch (MyBaseException exc)
    {
        cout << "Exception: " << exc.name() << endl;
    }
}
```



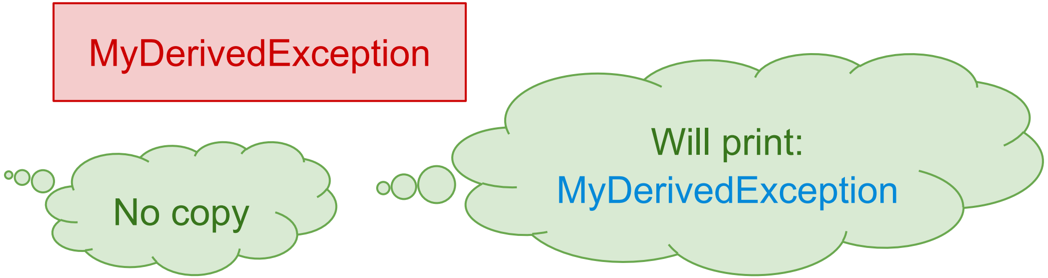
MyDerivedException

Slicing

MyBaseException exc

Will print: MyBaseException

```
void catchFunc()
{
    try
    {
        throw MyDerivedException();
    }
    catch (const MyBaseException& exc)
    {
        cout << "Exception: " << exc.name() << endl;
    }
}
```



MyDerivedException

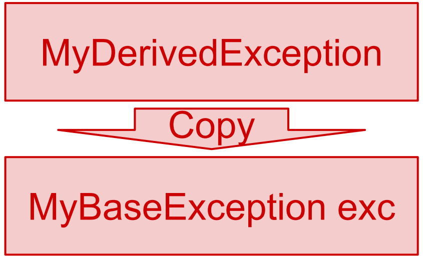
No copy

Will print: MyDerivedException

throw vs throw exc

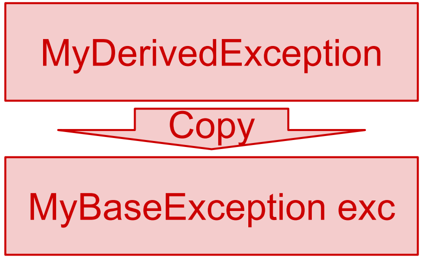
Normally you use “throw exc;” to throw.
Inside a catch block you can use “throw;” to re-throw current exception.

```
void catchFunc()
{
    try
    {
        throw MyDerivedException();
    }
    catch (MyBaseException exc)
    {
        throw exc;
    }
}
```



Will throw exc.
MyBaseException

```
void catchFunc()
{
    try
    {
        throw MyDerivedException();
    }
    catch (MyBaseException exc)
    {
        throw;
    }
}
```



Will throw original exception:
MyDerivedException

noexcept

C++ Core Guidelines
E.12

noexcept is used to mark functions that don’t throw exceptions.
If you will throw from such function std::terminate() will be called.

```
void lyingFunc() noexcept
{
    throw MyException();
}
```

std::terminate()

noexcept moves and optimisations

noexcept is used often to mark copy and move constructors, as well as std::swap() overloads.
Based on those decorations a more efficient implementation of some function can be called.
Example:

std::vector<T>::push_back() (must be strongly exception safe)

T has noexcept move constructor	push_back() will use moves	efficient
otherwise	push_back() will use copies	less efficient

C functions must not throw

C functions are not expected to throw.

If you want to pass a callback into a C api - don’t throw.

```
void bad_callback()
{
    throw 7;
}

void use_c_api()
{
    power_register_callback(&bad_callback, NULL);
}
```

```
int nicer_callback()
{
    try
    {
        possiblyThrow();
        return ESUCCESS;
    }
    catch (...)
    {
        return EFAIL;
    }
}
```

C:

- doesn’t know about exceptions,
- doesn’t have destructors,
- doesn’t have code for unwinding stack,
- might not even generate stack frames,
- etc...

So remember to catch.

Consistent exception handling

How can we refactor the error handling code to reduce duplication?

```
void doSomeWork()
{
    try
    {
        someWork();
    }
    catch (const NetworkError& exc)
    {
        loadFromFile();
    }
    catch (const InvalidData& exc)
    {
        dropConnection();
    }
    catch (const TooMuchData& exc)
    {
        reSendSmallerRequest();
    }
}
```

```
void doSomeOtherWork()
{
    try
    {
        otherWork();
        otherStuff();
    }
    catch (const NetworkError& exc)
    {
        loadFromFile();
    }
    catch (const InvalidData& exc)
    {
        dropConnection();
    }
    catch (const TooMuchData& exc)
    {
        reSendSmallerRequest();
    }
}
```

```
void doNothing()
{
    try
    {
        sleep();
        wait();
        sleep();
    }
    catch (const NetworkError& exc)
    {
        loadFromFile();
    }
    catch (const InvalidData& exc)
    {
        dropConnection();
    }
    catch (const TooMuchData& exc)
    {
        reSendSmallerRequest();
    }
}
```

Consistent exception handling

We can use Lippincott Functions (aka. exception dispatcher).

```
void doSomeWork()
{
    try
    {
        someWork();
    }
    catch (...)
    {
        handleExceptions();
    }
}
```

Catch any exception, and let exception handling function take care of it.

```
void doSomeOtherWork()
{
    try
    {
        otherWork();
        otherStuff();
    }
    catch (...)
    {
        handleExceptions();
    }
}
```

Current exception is a thread-local global, so it can be accessed inside handleExceptions().

```
void handleExceptions()
{
    try
    {
        throw;
    }
    catch (const NetworkError& exc)
    {
        loadFromFile();
    }
    catch (const InvalidData& exc)
    {
        dropConnection();
    }
    catch (const TooMuchData& exc)
    {
        reSendSmallerRequest();
    }
}
```

What to throw
and
when to catch

Exceptions in C++ standard library

All exceptions generated by the standard library inherit from std::exception.

There are two semantic classes of exceptions:

logic_error - when invariants are violated.

Your program has a bug.

runtime_error - failures caused by the environment.

I.e.when network failed.

And some commonly encountered exceptions:

bad_alloc - memory allocation failed.

bad_cast - dynamic_cast failed.

ios_base::failure - iostreams operation failed.

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error (C++11)
- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error (C++11)
  - tx_exception (TM TS)
  - system_error (C++11)
    - ios_base::failure (C++11)
    - filesystem::filesystem_error (C++17)
- bad_typeid
- bad_cast
  - bad_any_cast (C++17)
- bad_weak_ptr (C++11)
- bad_function_call (C++11)
- bad_alloc
  - bad_array_new_length (C++11)
- bad_exception
- ios_base::failure (until C++11)
- bad_variant_access (C++17)

What to throw?

It doesn’t really matter that much.
As you see, std::exception is not magic.
It’s just a very simple class.

```
class exception {
public:
    exception() noexcept;
    exception(const exception&) noexcept;
    exception& operator=(const exception&) noexcept;
    virtual ~exception();

    virtual const char* what() const noexcept;
};
```

You can derive from std::exception, logic_error or runtime_error.
But you can just as well write your own class.

What is important is to understand the basic principle:
Exception classes are semantic tags, that you can use to differentiate failure causes.

If you don’t need special handling for some error - just use std::runtime_error.

If you don’t need to handle two failure cases differently - don’t create different exceptions.

Guideline

C++ Core Guidelines E.14

- Start with throwing std::runtime_error.
  If you need to catch exceptions - create your own exception class.

- Create exception classes only when actually needed to solve a problem.
  - I need to show OpenFile dialog on FileNotFound error.
  - I need to show error message on FileNotFound error.

Create FileNotFound exception.

Just use std::runtime_error(“ File not found.”).

- Don’t create inheritance hierarchy, unless it actually serves a purpose.

```
MyException
    MyBackendException
        FileNotFoundInBackend
        BackendUnknownFailure
    MyApplicationException
        FileNotFoundApplication
        ApplicationUnknownFailure
```

Why?

```
NetworkException
    CurlException
    HttpException
        RangeHeaderIgnoredException
```

We generally need common handling.

But in some cases we need different handling.

When to catch?

C++ Core Guidelines
E.17 E.18

There is a simple guideline for this:

Don’t catch exceptions.

You should catch them only when you can do something meaningful with them.

Otherwise just let them fly to the top level.
On the top level report failure and continue or exit.

Top level is where you need to transform exception into something else, like DBus message or UI message.
It can also be module or API boundary.

In particular don’t catch exceptions just to throw a different one.

Example

```
void parseConfigFile(const std::string& fileName)
{
    std::string data;
    try
    {
        data = readFile(fileName);
    }
    catch (const FileNotFound& exc)
    {
        throw ParsingError("File not found.");
    }
    catch (const FileReadError& exc)
    {
        throw ParsingError("File read error.")
    }

    try
    {
        parse(data);
    }
    catch (const bad_alloc& exc)
    {
        throw ParsingError("Out of memory.")
    }
}
```

```
void parseConfigFile(const std::string& fileName)
{
    std::string data = readFile(fileName);
    parse(data);
}
```

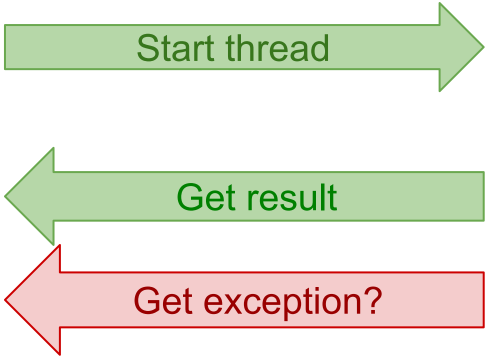
Just let the exceptions fly.

“Often the best way to deal with exceptions is to not handle them at all. If you can let them pass through your code and allow destructors to handle cleanup, your code will be cleaner.” David Abrahams

Exception propagation

Exception propagation

```
void main()
{
    startBackgroundWork();

    sleep(5000);

    int result;
    getBackgroundResult(result);
}
```



Start thread

Get result

Get exception?

```
void backgroundWork(int& result)
{
    if (random(0, 1) == 1)
        result = work();
    else
        throw std::exception("I'm lazy.");
}
```

```
void backgroundWork(int& result, T? exception)
{
    try
    {
        if (random(0, 1) == 1)
            result = work();
        else
            throw std::exception("I'm lazy.");
    }
    catch (...)
    {
        exception = ?;
    }
}
```

We need a way to capture an exception,
and pass it to the calling thread.

Problem: we don't know the type of the exception.
It can be anything.
So a simple parameter won't do.

Exception pointers

Fortunately there is a mechanism for storing any exception:

std::exception_ptr

```
void backgroundWork(int& result, std::exception_ptr& exception)
{
    try
    {
        if (random(0, 1) == 1)
            result = work();
        else
            throw std::exception("I'm lazy.");
    }
    catch (...)
    {
        exception = std::current_exception();
    }
}
```

std::exception_ptr is like a shared_ptr to a copy or reference of the current exception.

What can we do with std::exception_pointer?

Not much.

- We can copy it.
- We can create one from exception object: std::make_exception_ptr(MyException())
- We can check if it’s not null.
- But most importantly we can re-throw the underlying exception.

```
void main()
{
    startBackgroundWork();

    sleep(5000);

    int result;
    std::exception_ptr exc;

    getBackgroundResult(result, exc);

    if (exc)
        std::rethrow_exception(exc);
}
```

```
void backgroundWork(int& result, std::exception_ptr& exception)
{
    try
    {
        if (random(0, 1) == 1)
            result = work();
        else
            throw std::exception("I'm lazy.");
    }
    catch (...)
    {
        exception = std::current_exception();
    }
}
```

Similarly we can propagate from C callbacks.

Lippincott functions revisited

We can implement Lippincott functions using std::exception_pointer.

```
void doSomeWork()
{
    try
    {
        someWork();
    }
    catch (...)
    {
        auto exc = std::current_exception();
        handleExceptions(exc);
    }
}
```

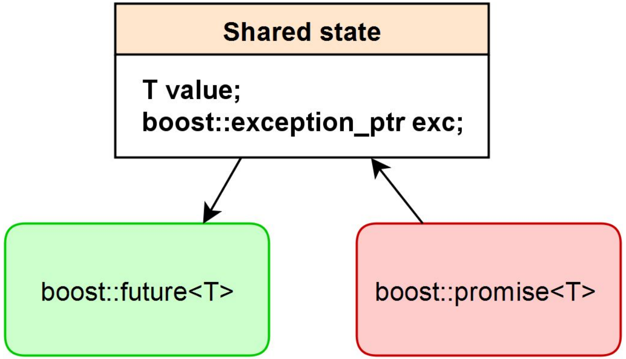
```
void handleExceptions(const std::exception_ptr& exc)
{
    try
    {
        if (exc)
            std::rethrow_exception(exc);
    }
    catch (const NetworkError& exc)
    {
        loadFromFile();
    }
    catch (const InvalidData& exc)
    {
        dropConnection();
    }
    catch (const TooMuchData& exc)
    {
        reSendSmallerRequest();
    }
}
```

Futures

std::future<T> is a class that can be used to wait for some background computation to finish.

It can be in the following states:

- waiting for result,
- holding a result,
- holding an exception thrown while computing the value in background.



Shared state

T value;
boost::exception_ptr exc;

boost::future<T>

boost::promise<T>

```
std::promise<int> promise;

void threadMethod()
{
    try
    {
        int result = computation();
        promise.set_value(result);
    }
    catch(...)
    {
        promise.set_exception(std::current_exception());
    }
}

void main()
{
    boost::thread thread(&threadMethod);
    boost::future<int> future = promise.get_future();

    // waits until computation ends...
    // ...then returns result or throws
    int result = future.get();

    thread.join();
}
```

Propagation through network, DBus, etc.

Often we want to propagate exceptions from another process, or another time:

- network connection,
- RPC,
- file storage (result serialization),
- database,
- different language,
- etc.

Exceptions can be arbitrary types.
There is no silver bullet.

- Serialize important exceptions.
- Pass the rest as generic exception.
  - Include full information about original exception as string.

Catch that doesn’t catch

Consider this exception hierarchy:

```
class ExceptionA : public std::exception
{};

class ExceptionB : public std::exception
{};

class ExceptionC : public ExceptionA, public ExceptionB
{};
```

ExceptionC has two sub-objects of type std::exception.

How will this work?

```
void main()
{
    try
    {
        throw ExceptionC();
    }
    catch(const std::exception& exc) {
        cout << "Std!" << endl;
    }
    catch(...) {
        cout << "Other!" << endl;
    }
}
```

Ambiguity. This catch will be ignored.

This catch clause will be used instead.

So use virtual inheritance for exceptions.

Adding information to exceptions

boost::exception

Nested exceptions

Sometimes it is useful to catch one exception, but throw another.

In that case, to avoid losing information about the original cause of the problem, we can store one exception in another.

```cpp
struct MyException
{
    MyException(const char* message, std::exception_ptr cause)
        : m_message(message)
        , m_cause(cause)
    {}

    const char* m_message;
    std::exception_ptr m_cause;
};
```

```cpp
void myFunction()
{
    try
    {
        doWork();
    }
    catch (...)
    {
        auto exc = std::current_exception();
        throw MyException("myFunction failed", exc);
    }
}
```

std::nested_exception

Fortunately there is no need to manually support nested exceptions.
Support for them is included in the C++ standard.

std::throw_with_nested(Exception())

Throws Exception with current exception stored within it.

std::rethrow_if_nested(exc)

Extracts nested exception, and throws it.

```
struct _InternalExc : Exception, std::nested_exception
{
     InternalExc(const Exception& exc)
        : Exception(exc)
        , nested_exception(std::current_exception())
    {}
};

void throw_with_nested(const Exception& exc)
{
    throw _InternalExc(exc);
}
```

```
void rethow_if_nested(const Exception& exc)
{
    auto nested = dynamic_cast<std::nested_exception*>(&exc);
    if (nested)
    {
        nested->rethrow_nested();
    }
}
```

Multiple inheritance trick
used to add exception_ptr
field to your exception class.

boost::exception

Boost exception is a library, that invented:

- exception_ptr,
- current_exception.

It has since became part of the C++ standard.

One feature however was not included in the standard:

- ability to attach arbitrary data to exceptions.

This powerful functionality can be leveraged by deriving your exceptions from boost::exception, like this:

```
#include <boost/exception/all.hpp>

class MyException : public virtual boost::exception, virtual public std::exception
{
    ...
};
```

Deriving from
std::exception is optional.

Attaching information to exceptions

To any type deriving from boost::exception you can attach arbitrary data, using error_info.
boost::exception is a container of error_info objects.

```
#include <boost/exception/all.hpp>
#include <boost/exception/errinfo_errno.hpp>

class MyException : public virtual boost::exception, public virtual std::exception
{};

void myFunction()
{
    int result = fddup(STDIN);
    if (result != 0)
    {
        throw MyException() << errinfo_errno(errno) << throw_file(__FILE__) << throw_line(__LINE__);
    }
}
```

Extracting information from exceptions

You can extract error_info data from boost::exception using get_error_info.

```
class MyException : public virtual boost::exception, public virtual std::exception
{};

void myFunction()
{
    int result = fddup(STDIN);
    if (result != 0)
        throw MyException() << errinfo_errno(errno) << throw_file(__FILE__) << throw_line(__LINE__);
}

void main()
{
    try
    {
        myFunction();
    }
    catch (const boost::exception& x)
    {
        const int* e = boost::get_error_info<errinfo_errno>(x);
        if (e)
            cout << "Errno was: " << *e << endl;
    }
}
```

Get pointer to specified error_info object.

Writing your own error_infos

errinfo_api_function
errinfo_at_line
errinfo_errno
errinfo_file_handle
errinfo_file_name
errinfo_file_open_mode
errinfo_nested_exception
errinfo_type_info_name

error_info is just:

- a tag,
- a value.

Writing your own error_infos is very simple:

```
#include <boost/exception/error_info.hpp>

typedef boost::error_info<struct tag_errno, int> errno_info;
typedef boost::error_info<struct tag_severity, int> severity_info;
typedef boost::error_info<struct tag_description, std::string> description_info;
```

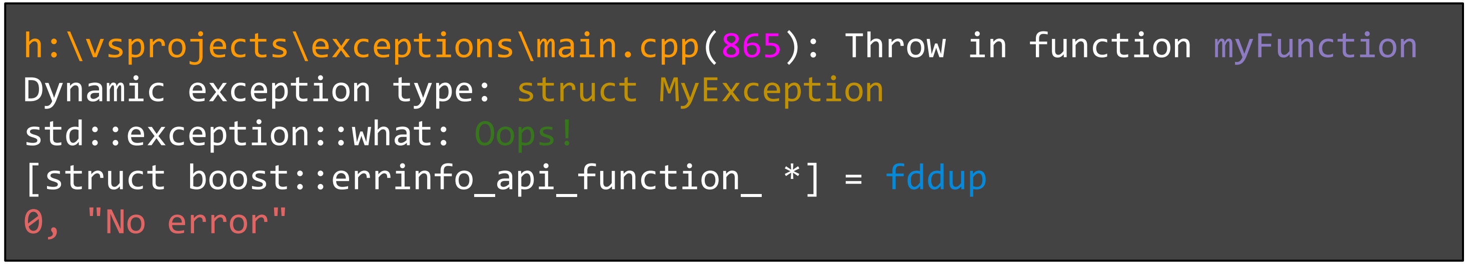
Usage:

```
BOOST_THROW_EXCEPTION(MyException("Oops!") << errno_info(errno) << description_info("Bad bug."));
```

current_exception_diagnostic_information

Instead of extracting all data by hand, you can use boost::current_exception_diagnostic_information() helper function, that will create a nice log message for you, with all the data.

```
struct MyException : virtual boost::exception, virtual std::exception {
    MyException(const char* msg) : std::exception(msg) {}
};

void myFunction()
{
    int result = fddup(STDIN);
    if (result != 0)
        throw MyException("Oops!") << errinfo errno(errno)
                                   << throw function("myFunction") << errinfo api function("fddup")
                                   << throw_file(__FILE__) << throw_line(__LINE__);
}

void main()
{
    try
    {
        myFunction();
    }
    catch (...)
    {
        cout << boost::
    }
}
```



```
h:\vsprojects\exceptions\main.cpp(865): Throw in function myFunction
Dynamic exception type: struct MyException
std::exception::what: Oops!
[struct boost::errinfo_api_function_ *] = fddup
0, "No error"
```

BOOST_THROW_EXCEPTION

BOOST_THROW_EXCEPTION is a helper macro, that:

- ensures, that boost::current_exception() works,
- automatically adds:
  - throw_function
  - throw_file
  - throw_line

```
void myFunction()
{
    int result = fddup(STDIN);
    if (result != 0)
        BOOST_THROW_EXCEPTION(MyException("Oops!") << errinfo_errno(errno) << errinfo_api_function("fddup"));
}
```

```
h:\exceptions\main.cpp(872): Throw in function void __cdecl myFunction(void)
Dynamic exception type: class boost::exception_detail::clone_impl<struct MyException>
std::exception::what: Oops!
[struct boost::errinfo_api_function_ *] = fddup
0, "No error"
```

Writing your own exception classes

Prefer empty classes, that inherit virtually from std::exception and boost::exception.
Attach all necessary data using error_infos.

```
struct MyException : public virtual boost::exception, public std::exception
{};
```

Inheriting from std::exception is a convention.

Don’t overload the meaning of exceptions from standard library.
Create your own exceptions for each purpose.

C++ Core
Guidelines
E.14

Thread interruption

How to stop background work?

Killing a thread is not an option.

- Memory leaks.
- Violated invariants.
- Locked mutexes.
- Etc.

Destructors are not called.

POSIX thread interruption is not an option.

- Doesn’t work with C++.

Destructors are not called.

Constantly checking the exitFlag?

- Manual work.
- Intrusive.
- Obfuscates the code.
- Requires modifications on all levels.
- Difficult to make sure you won’t block.

How often should we check the exitFlag?

Problem: passing of exitFlag

```
void backgroundWork(bool& exitFlag)
{
    while (!exitFlag)
    {
        waitForEvent();

        if (exitFlag)
            return;

        processData(exitFlag);

        if (exitFlag)
            return;

        doMore();
    }
}
```

```
void processData(bool& exitFlag)
{
    doStuff();

    if (exitFlag)
        return;

    doMoreStuff(exitFlag);
}
```

```
void doMoreStuff(bool& exitFlag)
{
    doStuff();

    if (exitFlag)
        return;

    doMore();
}
```

Solution: exitFlag should be a thread-global variable.

Problem: propagating cancel from bottom layers

```
void backgroundWork(bool& exitFlag)
{
    while (!exitFlag)
    {
        waitForEvent();

        if (exitFlag)
            return;

        result = calculateResult(exitFlag);

        if (exitFlag)
            return;

        doMore();
    }
}
```

```
float calculateResult(bool& exitFlag)
{
    for (int i = 0; i < 100000; ++i)
    {
        calculate();
        if (exitFlag)
            return -1.0f;
    }

    return 5.0f;
}
```

Magic number that means “cancelled”.

Solution: use exceptions to propagate the interruption.

Problem: when to check exitFlag

Usually exitFlag is checked when convenient:

- at the beginning of functions.
- once per loop iteration.
- etc.

When should exitFlag be checked?

During time consuming operations:

- while waiting for IO operation.
- while waiting for an event.
- while waiting for some time.
- during CPU intensive operations.

Normal exitFlag checks won’t help here.

Solution: boost interruption

Boost Thread library provides support for thread interruption. It consists of the following pieces:

- Each thread has a thread-global interruption flag.
- Each waiting function will throw boost::interrupted exception as soon as the interruption flag is set:

  boost::thread::join()
  boost::condition_variable::wait()
  boost::thread::sleep()
  etc...

- One can add manual check, which will throw boost::interrupted exception if the flag is set:

  boost::this_thread::interruption_point()

- Function for setting interruption flag of a thread:

  boost::thread::interrupt()

Example: before

```
void backgroundWork(bool& exitFlag)
{
    while (!exitFlag)
    {
        waitForEvent();

        if (exitFlag)
            return;

        result = calculateResult(exitFlag);

        if (exitFlag)
            return;

        doMore();
    }
}
```

```
float calculateResult(bool& exitFlag)
{
    for (int i = 0; i < 100000; ++i)
    {
        calculate();
        if (exitFlag)
            return -1.0f;
    }

    return 5.0f;
}
```

```
void main()
{
    bool exitFlag = false;
    boost::thread t(&backgroundWork, exitFlag);

    exitFlag = true;

    t.join();
}
```

UB

Example: after

```
void backgroundWork()
{
    while (true)
    {
        waitForEvent();

        result = calculateResult();

        doMore();
    }
}
```

```
float calculateResult()
{
    for (int i = 0; i < 100000; ++i)
    {
        calculate();
        boost::interruption_point();
    }

    return 5.0f;
}
```

Manual check.

```
void main()
{
    boost::thread t(&backgroundWork);

    t.interrupt();

    t.join();
}
```

- No exitFlag.
- Manual checks are rare.
- Almost no extra work is necessary.
- Code is interruptible by default.

Interrupting destructors

Destructors cannot throw.

Extra care needs to be taken, when writing destructors with interruptible functions inside.

```
struct Worker
{
    boost::thread t;

    Worker()
        : t(&workFunction)
    {}

    ~Worker()
    {
        t.join();
    }
};
```

```
struct Worker
{
    boost::thread t;

    Worker()
        : t(&workFunction)
    {}

    ~Worker()
    {
        boost::disable_interruption di;
        t.join();
    }
};
```

Interrupting threads

No exception is allowed to fly from the thread function.
So catch, and propagate.

```
void backgroundWork()
{
    try
    {
        while (true)
        {
            waitForEvent();

            result = calculateResult();

            doMore();
        }
       promise.set_value(result);
    }
    catch (const boost::thread_interrupted&)
    {
        promise.set_exception(Cancelled());
    }
    catch (...)
    {
        promise.set_exception(std::current_exception());
    }
}
```

More technicalities

Why C++ doesn’t have finally?

Because we have destructors.
Ad-hoc cleanup is bad. Use RAII.

ON_SUCCESS, ON_FAILURE

C++ Core Guidelines E.19

Generic Scope Guard p0052r7

Scope guards are used for unconditional cleanup.

```
 void usingC()
 {
     void* obj = gst_alloc_obj();
     SCOPE_EXIT(gst_free_obj(obj));

     doStuff(obj);
 }
```

Andrei Alexandrescu proposed two more kinds of scope guards:

- ON_SUCCESS will execute code if function exits normally.
- ON_FAILURE will execute code if function exits because of an exception.

```
void usingDatabase()
{
    auto t = startDatabaseTransaction();
    ON_SUCCESS(t.commit());
    ON_FAILURE(t.rolback());

    t.insert(stuff);
    t.remove(others)
}
```

noexcept destructors by default

This is a breaking change in C++ 11 standard.

It’s a bad idea to throw from destructors.

That’s why in C++ 11 destructors are by default noexcept.

- Unless they call a base or member destructor, that is noexcept(false).

If you want to throw from a destructor, you have to mark it as noexcept(false).

noexcept operator

C++ 11 adds a noexcept operator.
It can be used to conditionally execute code, or to more precisely define noexcept specifications.

```
template<typename T>
const char* getNameSafe(const T& object) noexcept
{
    if (noexcept(object.name()))
        return object.name();

    return "Name can throw."
}
```

```
template<typename T>
T maybeTrow() noexcept(sizeof(T) < 4)
{
    ...
}
```

```
template<typename T>
class MyValue
{
    void setDefault() noexcept(noexcept(T()))
    {
        std::swap(v, T());
    }

    void set(const T& item)
        noexcept(std::is_nothrow_copy_assignable<T>::value)
    {
        v = item;
    }

    T v;
};
```

Polymorphic throw

“throw e” statement throws an object with the same type as the static type of the expression e.

```
void doThrow(MyExceptionBase& e)
{
    throw e;
}

void throwAndCatch()
{
    MyExceptionDerived e;
    try
    {
        doThrow(e);
    }
    catch (MyExceptionDerived& e)
    {
        cout << "MyExceptionDerived.";
    }
    catch (...)
    {
        cout << "Something else.";
    }
}
```

Static type is MyExceptionBase.

This will match.

Why pop_back() returns void?

Guru of the Week 8

Vector interface:

```
void push_back(const T& v);
void pop_back();
```

Why not?

```
void push_back(const T& v);
T pop_back();
```

Consider the implementation:

```
T pop_back()
{
  if(empty())
    throw "Empty!";

  size--;

  return v[size];
}
```

What if T's copy constructor throws?

Not possible to achieve strong exception safety in this case.

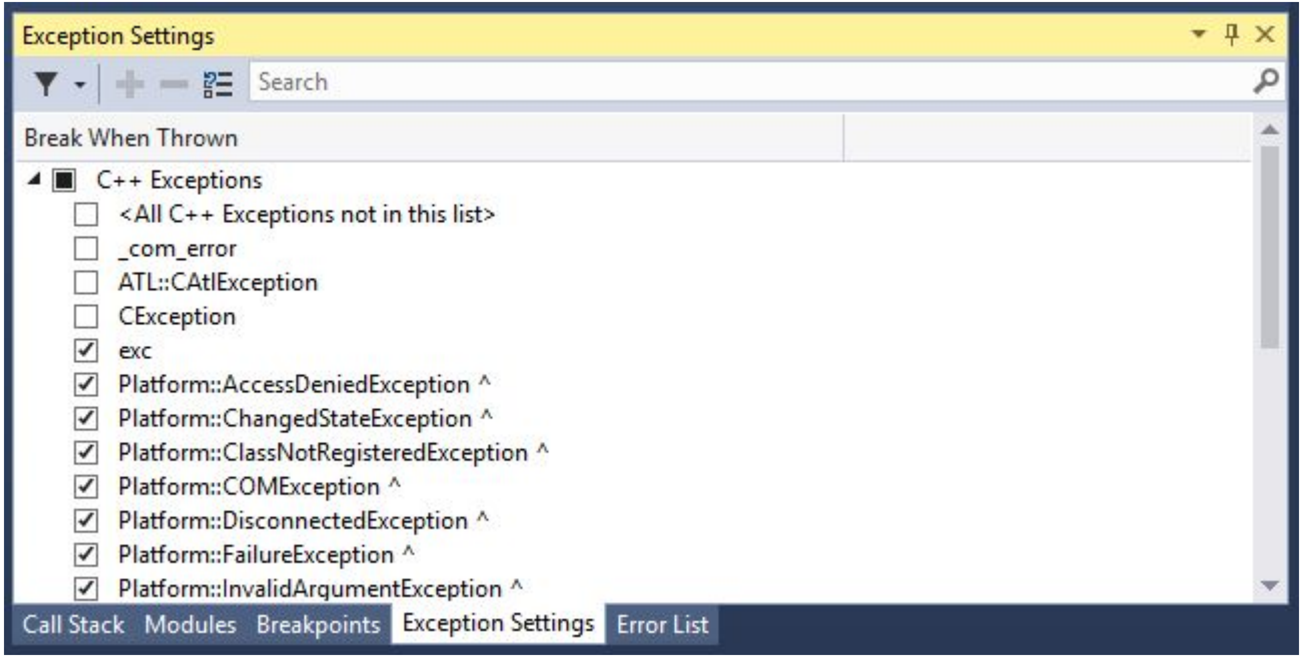
Function try blocks

Guru of
the Week
66

```
// Example 1(a): Constructor function-try-block
//
C::C()
try
  : A ( /*...*/ ) // optional initialization list
  , b_( /*...*/ )
{
}
catch( ... )
{
  // We get here if either A::A() or B::B() throws.

  // If A::A() succeeds and then B::B() throws, the
  // language guarantees that A::~A() will be called
  // to destroy the already-created A base subobject
  // before control reaches this catch block.
}
```

Breakpoints on throw

Debuggers support setting breakpoints on throw of given exception type.



Break When Thrown

- ■ C++ Exceptions
  - □ <All C++ Exceptions not in this list>
  - □ _com_error
  - □ ATL::CAtlException
  - □ CException
  - ☑ exc
  - ☑ Platform::AccessDeniedException ^
  - ☑ Platform::ChangedStateException ^
  - ☑ Platform::ClassNotRegisteredException ^
  - ☑ Platform::COMException ^
  - ☑ Platform::DisconnectedException ^
  - ☑ Platform::FailureException ^
  - ☑ Platform::InvalidArgumentException ^

Stack traces

Exceptions in C++ don’t have stacktraces.

All the unwinding machinery is there (in some implementations), but there is no way to access it.
Maybe in the future we will get standard way of getting stack traces.

- You can always use libunwind or StackWalker,
- And add stacktrace to your boost::exception as another error_info.

The End

C++ EXCEPTION HANDLING

The gory details of an implementation

Peter Edwards, Arista Networks

ARISTA

Dublin C/C++ Meetup, February 2018

Video: CppCon 2014: Exception-Safe Code, Jon Kalb
https://youtu.be/W7fly_54y-w

Exceptions and Error Handling FAQ, C++ Standards Committee
https://isocpp.org/wiki/faq/exceptions

Video: Systematic Error Handling in C++, Andrei Alexandrescu
https://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Andrei-Alexandrescu-Systematic-Error-Handling-in-C

Boost Exception Tutorial:
http://www.boost.org/doc/libs/1_61_0/libs/exception/doc/boost-exception.html

C++ Core Guidelines, Bjarne Stroustrup and Herb Sutter
https://github.com/isocpp/CppCoreGuidelines

Video: C++ Exception Handling - The gory details of an implementation, Peter Edwards
https://www.youtube.com/watch?v=XpRL7exdFL8

Video: C++ Exceptions and Stack Unwinding, Dave Watson
https://www.youtube.com/watch?v=_Ivd3qzgT7U

Interrupt Politely, Herb Sutter
http://www.drdobbs.com/parallel/interrupt-politely/207100682

Change the Way You Write Exception-Safe Code - Forever, Andrei Alexandrescu and Petru Marginean
http://www.drdobbs.com/cpp/generic-change-the-way-you-write-excepti/184403758

Exception Safety, Herb Sutter
http://www.gotw.ca/gotw/008.htm

Exception-Safe Class Design, Herb Sutter
http://www.gotw.ca/gotw/059.htm