Department I - C Plus Plus

Modern and Lucid C++ for Professional Programmers

Week 4 – Functions and Exceptions

Prof. Peter Sommerlad / Thomas Corbat

Rapperswil, 09.10.2018

HS2018

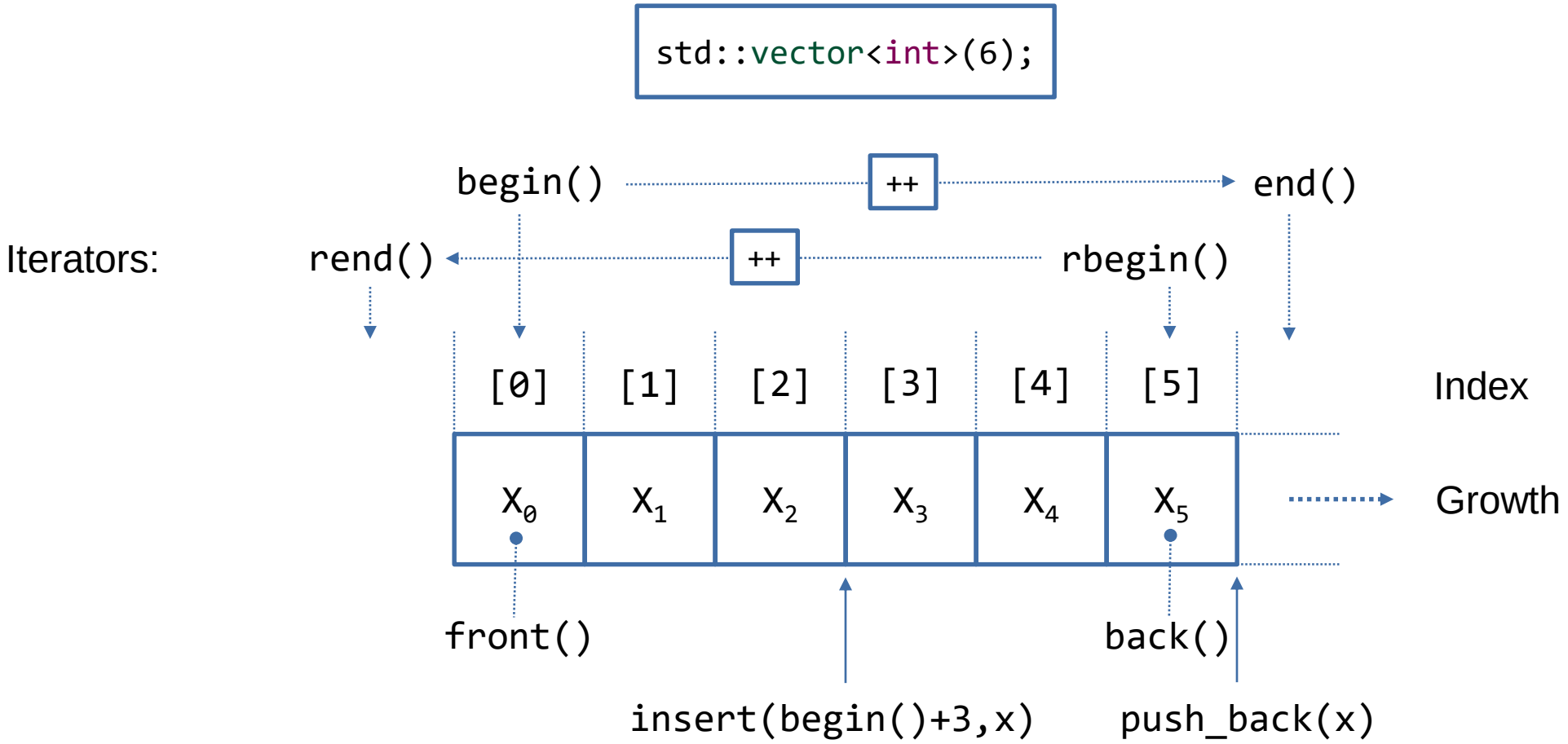
HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

Recap Week 3

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE



- **Parenthesis at definition allow providing initial size, when type of elements is a number**
  - `std::vector<std::string> words{6};` works

- **If possible always prefer a standard algorithm over a handwritten loop**

```
std::for_each(std::begin(container), std::end(container), action);
```

- **Shorter and okay if no corresponding algorithm exists ranged-base for**

```
for (auto entry : container)
```

- **Use iterator-based for loops when the position is required**

```
for (auto it = std::begin(container); std::end(container); it++)
```

- **Use index-based for loops only when the index is required**

```
for (size_t index = 0; index < container.size(); index++)
```

- **If possible always prefer a standard algorithm over a handwritten loop**
- **Specifying a (sub)range of elements in a container requires iterators**

```
std::vector<int> reverseSquared(std::vector<int> values) {
  auto const square = [](auto & v) {
    v *= v;
  };
  std::for_each(std::begin(values), std::end(values), square);
  std::vector reversedValues{std::crbegin(values), std::crend(values)};
  return reversedValues;
}
```

- **Standard output mimicking standard input**

```
void redirectStreamIO(std::istream & inputStream, std::ostream & outputStream) {
  using input = std::istream_iterator<char>;
  input eof{};
  input in{inputStream};
  std::copy(in, eof, std::ostream_iterator<char>{outputStream});
}
```

- **What happens to whitespace?**

```
void redirectStreamIO(std::istream & inputStream, std::ostream & outputStream) {
  using input = std::istream_iterator<char>;
  input eof{};
  input in{inputStream};
  inputStream >> std::noskipws;
  std::copy(in, eof, std::ostream_iterator<char>{outputStream});
}
```

Functions (One More Time)

Goals:

- You know the different ways of passing parameters to functions
- You know how to pass default arguments to functions
- You know how to write a lambda

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

	const: • Parameter cannot be changed	**non-const:** • Parameter can be changed

reference:
• Argument on call-site is accessed
	`void f(std::string const & s) {`
	`  //no modification`
	`  //efficient for large objects`
	`}`
		`void f(std::string & s) {`
		`  //modification possible`
		`  //side-effect also at call-site`
		`}`

copy:
• Function has its own copy of the parameter
	`void f(std::string const s) {`
	`  //no modification`
	`  //used for maximum constness`
	`}`
		`void f(std::string s) {`
		`  //modification possible`
		`  //side-effect only locally`
		`}`

- **Call-site always looks the same**
  - You cannot pass a const argument to a non-const reference

```
std::string name{"John"};
f(name);
```

- **Value Parameter:** `void f(type par);`
  - Default for parameters
- **Reference Parameter:** `void f(type & par);`
  - When side-effect is required at call-site
- **Const-Reference Parameter:** `void f(type const & par);`
  - Possible optimization, when type is large (costly to copy) and no side-effects desired at call-site
  - For non-copyable objects
- **Const Value Parameter:** `void f(type const par);`
  - The coding style guide of your project this might prefer this over non-const value parameters
  - Could prevent changing the parameter in the function inadvertently

- **By Value:** `type f();`
  - Default for return types
- **By Reference:** `type & f(); or type const & f();`
  - Only return a reference parameter (or a class member variable from a member function)

```
std::ostream & sayHello(std::ostream & out) {
  return out << "Hello";
}
```

  - Never return a reference to a local variable!

```
std::string & create() {
  std::string name{"John"};
  return name;
}
```

- **By Const Value:** `type const f();`
  - Don't do this, it just annoys the caller


```
Connection & connect(Address address) {
  Connection connection{address};
  //...
  return connection;
}
```

Incorrect

The connection only lives within the connection function. The returned reference will be dangling. A compiler can detect and report this.

```
POI createPOI(Coordinate);

auto allPOIs(Corrdinate const location) {
  //...
  POI const & migros = createPOI(location);
  //...
  return std::vector{migros};
}
```

Unusual, but correct

`const &` extends the life-time of the temporary POI, until the end of the block. The POI will be copied into the returned `std::vector` object

```
void modify(LargeDocument & document);

void changeDocument() {
  LargeDocument const document{};
  modify(document);
}
```

Incorrect

The document object is const, therefore it must not be modified (directly or indirectly). The function modify might change it as it takes a LargeDocument by reference.

Right or Wrong?


```
void print(LargeDocument const & document);

void printAll() {
  LargeDocument document{};
  print (document);
}
```

Correct

The (large) object document can be passed as `const &`, even though it is modifyable.

```
std::string const & max(
    std::string const & left,
    std::string const & right) {
  return (left > right) ? left : right;
}

int main() {
  std::string const & larger = max("a", "b");
  std::cout << "larger is: " << larger;
}
```

Incorrect

`const &` only extends the life-time of temporaries. The return type of `max()` is a reference, not a value. Therefore, its life-time is bound to the life-time of the returned object. The returned object is either argument of `max()`. Those in turn are temporary std::string objects created from "a" and "b", which only live until the end of the statement. Beware: Small string optimization!

```
void incr(int & var);
void incr(int & var, unsigned delta);
```

- **The same function name can be used for different functions if parameter number or types differ**
  - Functions cannot be overloaded just by their return type
  - If only the parameter type is different there might be ambiguities
- **Resolution of overloads happens at compile-time = Ad hoc polymorphism**
- **Internal name of a function also contains its parameter types as significant information (not the parameter names)**
  - Information required by the linker

```
int factorial(int n) {
  if (n > 1) {
    return n * factorial(n - 1);
  }
  return 1;
}

double factorial(double n) {
  double result = 1;
  if (n < 15) {
    return factorial(static_cast<int>(n));
  }
  while (n > 1) {
    result *= n;
    --n;
  }
  return result;
}
```

```
void demoAmbiguity() {
  std::cout << factorial(3) << '\n';
  std::cout << factorial(1e2) << '\n';

  std::cout << factorial(10u) << '\n';
  std::cout << factorial(1e1L) << '\n';
}
```

```
void incr(int & var, unsigned delta = 1);
```

- **A function declaration can provide default arguments for its parameters from the right**
  - Definition doesn't need to/shouldn't repeat
  - = is required as part of the parameter declaration

```
void incr(int & var, unsigned delta) {
  var += delta;
}
```

- **Implicit overload of the function with fewer parameters**
  - If n default arguments are provided, n+1 versions of the function are declared
- **Default arguments can be omitted when calling the function**

```
int counter{0};
incr(counter);     //uses default for delta
incr(counter, 5);
```

```
void applyAndPrint(double x, double f(double)) {
  std::cout << "f(" << x << ") = " << f(x) << '\n';
}
```

- **Functions are "first class" objects in C++**
  - You can pass them as argument
  - You can keep them in reference variables

```
double (&h)(double)
```

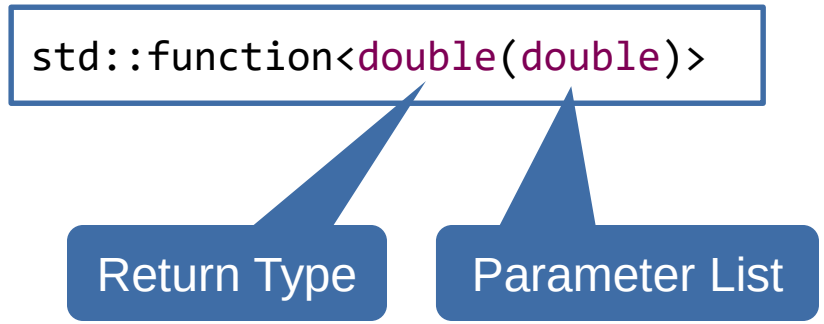
- **Drawback: A function parameter declared in this way does not accept a lambda <u>with a capture</u>**

```
int main() {
  double factor{3.0};
  auto const multiply = [factor](double value) {
    return factor * value;
  };
  applyAndPrint(1.5, multiply);
}
```

- **Modern C++ approach: `std::function` template, which also allows passing lambdas (with capture)**

```
void applyAndPrint(double x, std::function<double(double)> f) {
  std::cout << "f(" << x << ") = " << f(x) << '\n';
}

int main() {
  double factor{3.0};
  auto const multiply = [factor](double value) {
    return factor * value;
  };
  applyAndPrint(1.5, multiply);
}
```



?

```
std::vector<int> intsUpTo(
                        std::size_t from = 0,
                        std::size_t to) {
  //ensure to >= from
  std::vector<int> values(to - from + 1);
  iota(begin(values), end(values), from);
  return values;
}
```

Incorrect

Default arguments have to be specified from right to left. If the definition and the declaration are separated the default argument should be specified at the declaration in the header.

```
void square(int x, void print(int)) {
  print(x * x);
}

int main() {
  auto p = [](auto value) {
    std::cout << value;
  };
  square(5, p);
}
```

Legacy, but correct

As the lambda does not capture anything it can be passed as function argument as is. However, a modern approach would use `std::function<void(int)>` as parameter type for the print function. E.g. a lambda capturing a stream could not be passed otherwise.

- **Defining inline functions**

```
auto const g = [](char c) -> char {
  return std::toupper(c);
};
g('a');
```

- **`auto const` for function variable from Lambda**
- **`[]` introduces a Lambda function**
  - Can contain captures: `[=]` or `[&]` to access variables from scope
- **`(Parameters)` as with other functions, but optional if empty**
  - Parameters can be `auto` (Generic Lambda)
- **Body block with statements**

- **Capturing a local variable by value**
  - Local copy lives as long as the lambda lives
  - Local copy is immutable, unless lambda is declared `mutable`

```
int x = 5;
auto l = [x]() mutable {
  std::cout << ++x;
};
```

- **Capturing a local variable by reference**
  - Allows modification of the captured variable
  - Side-effect is visible in the surrounding scope, but referenced variable must live at least as long as the lambda lives

```
int x = 5;
auto const l = [&x]() {
  std::cout << ++x;
};
```

- **Capturing all local variables by value or reference**

```
int x = 5;
auto l = [=]() mutable {
  std::cout << ++x;
};
```

```
int x = 5;
auto const l = [&]() {
  std::cout << ++x;
};
```

(More) Lambda Capture Examples

- **Capturing `this` pointer**
  - Allows accessing and modifying members of the class

```
struct S {
  void foo() {
    auto square = [this] {
      member *= 2;
    };
  }
private:
  int member{};
};
```

- **New local variable can be specified in capture**
  - New variable in capture has type `auto`
  - Can be modified if lambda is `mutable`

```
auto squares = [x=1]() mutable {
  std::cout << x *= 2;
};
```

- **In captures multiple variables can be combined and separated with commas (,)**

- **In function definitions the return type can be declared auto**
- **The actual return type will be deduced from the return statements in the function's body**
  - The body must be present!

```
auto middle(std::vector<int> const & c) {
  //check not empty
  return c[c.size() / 2];
}
```

```
auto <function-name>(<parameters>) -> <return-type>;
```

- **If the return type of a function is declared as auto a trailing return type can specify the return type**
  - In this case the function body is not required when specifying a trailing return type

```
auto middle(std::vector<int> const & c) -> int;

auto middle(std::vector<int> const & c) -> int {
  //check not empty
  return c[c.size() / 2];
}
```

  - Can be used to explicitly specify the return type of a lambda

```
auto isOdd = [](auto value) -> bool {
  return value % 2;
};
```

?

```
#include <iostream>

int calculateAnswer();

int main() {
  std::cout << calculateAnswer();
}
```

Correct

As long as the function is defined in another compilation unit (.cpp file). `main()` does not need a `return` statement. It implicitly returns `0` if none is provided.
It is unsual to explicitly provide a forward declaration of a function. A function usually is declared in a header file.

```
#include <iostream>

auto maxValue(int f, int s, int t);

int main() {
  std::cout << maxValue(1, 2, 3);
}
```

Incorrect

The compiler needs a function body in order to deduce the retrun type. It would also be possible to insert a function definition before its first use or to supply a trailing return type instead. The latter would be unusual in this case.

Failing Functions

Goals:

- You know 5 different ways to react to errors in functions
- You know how to throw, catch and test exceptions

- **Precondition (Assumption) is violated**
  - Negative index
  - Divisor is zero
  - Usually caller provided wrong arguments
- **A function without preconditions has a so-called "wide contract" as opposed to "narrow contract"**
- **Postcondition could not be satisfied**
  - Resources for computation are not available
  - Can not open a file

Contract cannot be fulfilled

- **What should you do, if a function cannot fulfill its purpose?**
  1. Ignore the error and provide potentially **undefined behavior**
  2. Return a **standard result** to cover the error
  3. Return an **error code** or error value
  4. Provide an **error status** as a side-effect
  5. Throw an **exception**
- **But first you need to know, if it can fail at all!**

```
std::vector v{1, 2, 3, 4, 5};
v[5] = 7;
```

- **Relies on the caller to satisfy all preconditions**
- **Viable only if not dependent on other resources**
- **Most efficient implementation**
  - No unnecessary checks
- **Simple for the implementer but harder for the caller**
- **Should be done consciously and consistently!**

```
std::string inputName(std::istream & in) {
  std::string name{};
  in >> name;
  return name.size() ? name : "anonymous";
}
```

- **Reliefs the caller from the need to care if it can continue with the default value**
- **Can hide underlying problems**
  - Debugging can give you nightmares
- **Often better if caller can specify its own default value**

```
std::string inputNameWithDefault(std::istream & in,
                                 std::string const & def = "anonymous") {
  std::string name{};
  in >> name;
  return name.size() ? name : def;
}
```

- **Only feasible if result domain is smaller than return type**
  - There exists a value that can be used
  - Sometimes invented artificially: `std::string::npos`

```
bool contains(std::string const & s, int number) {
  auto substring = std::to_string(number);
  return s.find(substring) != std::string::npos;
}
```

  - POSIX defines -1 to mark failure of many system calls
- **Burden on the caller to check the result**
  - Danger of ignoring significant errors if result is otherwise insignificant

- **Encodes the possibility of failure in the type system**
  - Can optionally contain NO value (default construction)

```
std::optional<std::string> inputName(std::istream & in) {
  std::string name{};
  if (in >> name) return name;
  return {};
}
```

- **Requires explicit accessof the value at the call site**
  - `has_value()` or boolean conversion checks whether the optional contains a value

```
int main() {
  std::optional name = inputName(std::cin);
  if (name.has_value()) {
    std::cout << "Name: " << name.value() << '\n';
  }
}
```

```
int main() {
  std::optional name = inputName(std::cin);
  if (name) {
    std::cout << "Name: " << *name << '\n';
  }
}
```

- **Requires reference parameter**
  - Can be `this` object in member functions
  - Annoying when error variable must be provided

```
int connect(std::string url, bool & error) {
  //set error when an error occurred
}
```

- **(Bad!) Alternative: Global variable**
  - POSIX' `errno` is the glorious example of that
- **Example: `std::istream`'s state (`good()`, `fail()`) is changed as a side-effect of input**

```
std::string name{};
in >> name;
if (in.fail()) { //Member variable
  //Handle error case
}
```

```
throw std::invalid_argument{"Description"};
throw 15;
```

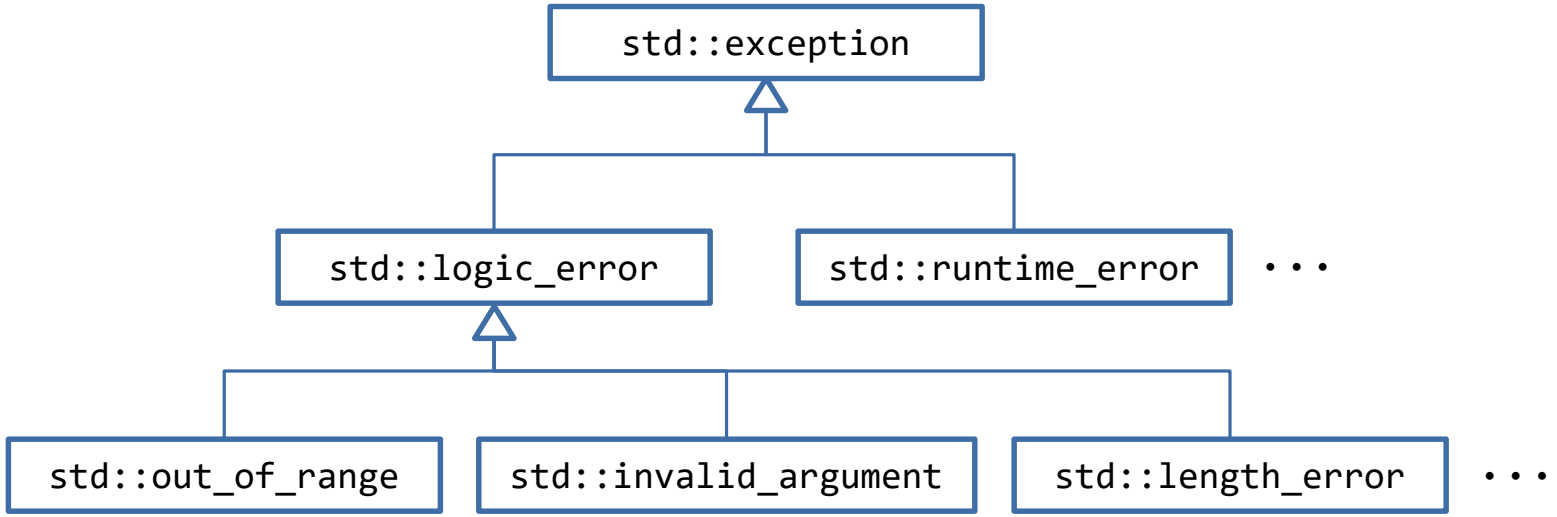
- **To throw an exception:** `throw value;`
  - Any (copyable) type can be thrown
- **No means to specify what could be thrown**
  - No checks if you catch an exception that might be thrown at call-site
- **No meta-information is available as part of the exception**
  - No stack trace, no source position of throw
- **Exception thrown while exception is propagated results in program abort (not while caught)**

- **Try-catch block as in Java**
  - Can be the whole function body
- **Principle: Throw by value, catch by const reference**
  - Avoids unnecessary copying
  - Allows dynamic polymorphism for class types
- **Sequence of catches is significant**
  - First match wins
- **Catch all with ellipsis (...):**
  - Must be last catch
- **Caught exceptions can be rethrown with `throw;`**

```
try {
  throwingCall();
} catch (type const & e) {
  //Handle type exception
} catch (type2 const & e) {
  //Handle type2 exception
} catch (...) {
  //Handle other exception types
}
```

`#include <stdexcept>`

- **The Standard Library has some pre-defined exception types that you can also use in <stdexcept>**



- **All have a constructor parameter for the "reason" of type `std::string`**
- **`std::exception` is the base class**
  - It provides the `what()` member function to obtain the "reason"

- **Functions that have a precondition on their caller**
  - When not all possible argument values are useful for the function

```
double square_root(double x) {
  if (x < 0) {
    throw std::invalid_argument{"square_root imaginary"};
  }
  return std::sqrt(x);
}
```

- **Do NOT use exceptions as a second means to return values**
  - Catch then becomes a "come from" and throw a "go to"

- **CUTE provides ASSERT_THROWS(code, exception)**

```
void testSquareRootNegativeThrows() {
  ASSERT_THROWS(square_root(-1.0), std::invalid_argument);
}
```

```
void testEmptyVectorAtThrows() {
  std::vector<int> empty_vector{};
  ASSERT_THROWS(empty_vector.at(0), std::out_of_range);
}
```

- **You can also use try-FAILM()-catch**

```
void testForExceptionTryCatch() {
  std::vector<int> empty_vector{};
  try {
    empty_vector.at(1);
    FAILM("expected Exception");
  } catch (std::out_of_range const &) {
    // expected
  }
}
```

?

```
void check(int i) {
  if (i % 2) {
    throw "is even";
  throw 0;
}

void printIsEven(int i) try {
  check(i);
} catch(int) {
  std::cout << "that's odd";
} catch(...) {
  std::cout << "very even";
}
```

Incorrect

It is syntactically correct C++, BUT:

- You must never use an exception on correct execution paths just to change control flow on the call-site. Use exceptions only for violations of pre- and post-conditions!
- Don't throw primitives. Use an exception from <stdexcept> or derive your own exception from std::exception.
- Catch by const reference, not by value!
- Condition is inverted

- **A good function**
  - Does one thing well and is named after that ("High Cohesion")
  - Has only few parameters (<=3, max 5)
  - Consists of only a few lines without deeply nested control structure
  - Provides guarantees about its result (aka its contract)
  - Is easy to use with all possible argument values its parameter types allow or provides consistent error reporting if argument values prohibit delivering its result (exception)
- **Pass parameters and return results by value (unless there is a good reason not to)**