

7. Error handling

Sven Gestegård Robertz  
Computer Science, LTH

2018



Outline

- 1 Error handling
  - Exceptions
  - Catching exceptions
  - Throwing exceptions
  - Exceptions and resource management
  - Specification of exceptions
  - Static assert
  - assert
- 2 type casts

Error handling

Three levels of error handling

- 1 Directly handle the error locally and continue execution
- 2 Categorize and pass error to another module that is expected to handle it
- 3 Identify the error, give an error message, and crash the program (*“fail-fast”, e.g., assert*)

Level 2: exceptions (or return values)

Throwing exceptions

Example: checking arguments in the Vector class

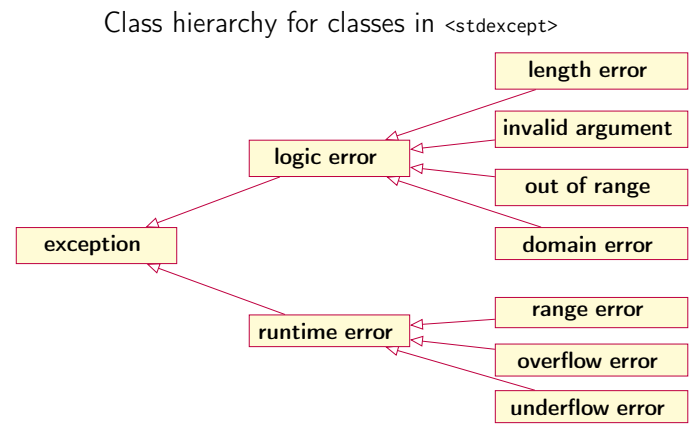
```
Vector::Vector(int size) {  
    if(size < 0) throw length_error("negative size");  
    elem = new double[size];  
    sz = size;  
}  
  
int& Vector::operator[] (int i) {  
    if (i<0 || i>=sz) throw out_of_range("Vector::operator[]");  
    return elem[i];  
}
```

- ▶ NB: to allow checking arguments, we use a signed integer type for values that should always be positive
- ▶ Vector cannot reasonably handle the error locally, only the caller can know why it passed a certain argument

Exceptions

- ▶ Error handling is done with `throw` and `catch`. Like Java.
- ▶ *“stack unwinding”* until a matching `catch` is found.
- ▶ When an exception is thrown, activation records are popped off the stack until a function containing a matching `catch` is found. (*“stack unwinding”*)
- ▶ If an exception is not caught, the program crashes. (by calling `std::terminate()`.)
- ▶ Standard classes for exceptions: `#include <stdexcept>`

The exception classes of the standard library



## Error handling

### Catching exceptions

```
try {
    // Code that may throw
}
catch (some_exception&) {
    // Code handling some_exception
}
catch (another_exception&) {
    // Code handling another_exception
}
catch (...) {
    // default/generic exception handling
}
```

The first `catch` clause with a matching type is selected.  
⇒ Catch derived classes before the base class.

... is valid C++, matches anything

## Catching exceptions

### Example:

```
try {
    cout << "Enter a number: ";
    int i;
    if (cin >> i) {
        int r = f(i);
        cout << "Result: " << r << endl;
    }
}
catch(std::overflow_error&) {
    cout << "Overflow error\n";
}
catch(std::exception& e) {
    cout << typeid(e).name() << ": " << e.what() << endl;
}
```

## Catching exceptions

### Exempel:

```
try {
    cout << "Enter a number: ";
    int i;
    if (cin >> i) {
        int r = f(i);
        cout << "Result: " << r << endl;
    }
}
catch(std::overflow_error&) {
    cout << "Overflow error\n";
}
catch(std::exception& e) {
    cout << typeid(e).name() << ": " << e.what() << endl;
}
```

predefined function in the class exception

## Catching exceptions

### ... and rethrowing

```
try{
    do_something();
}
catch {std::length_error& le) {
    // handle length error
}
catch {std::out_of_range&) {
    // handle out_of_range
}
catch (...) {
    throw; // default: pass on
}
```

## Throwing exceptions

### Creating own exceptions as subclasses

```
#include <stdexcept>

class communication_error : public runtime_error {
public:
    communication_error(const string& mess = "")
        : runtime_error(mess) {}
};
```

### Throwing

```
throw communication_error("Checksum error");
```

## Throwing exceptions

### Creating custom exceptions

```
struct MyOwnException{
    MyOwnException(const std::string& msg, int val)
        : m(msg),v{val} {}
    std::string m;
    int v;
};
```

### Using custom exceptions

```
void f() {
    throw MyOwnException("An error occurred", 17);
}

void test1() {
    try{
        f();
    } catch(MyOwnException &e) {
        cout << "Exception: " << e.m << " - " << e.v << endl;
    }
}
```

## Catching exceptions

Resource mangement: destructors and "stack unwinding"

```
struct Foo {
    int x;
    Foo(int ix) :x{ix} {
        cout << "Foo("<<x<<")\n";
    }
    ~Foo() {
        cout << "~Foo("<<x<<")\n";
    }
};

void test(int i)
{
    Foo f(i);
    if(i == 0) {
        throw std::out_of_range("noll?");
    } else {
        Foo g(100+i);
        test(i-1);
        cout << "after call to test("
            << i-1 << ")\n";
    }
}
```

```
int main() {
    Foo f(42);
    try{
        Foo g(17);
        test(2);
    } catch(std::exception& e) {
        cout<<e.what()<< endl; }
}
    Foo(42)
    Foo(17)
    Foo(2)
    Foo(102)
    Foo(1)
    Foo(101)
    Foo(0)
    ~Foo(0)
    ~Foo(101)
    ~Foo(1)
    ~Foo(102)
    ~Foo(2)
    ~Foo(17)
    ~Foo(42)
```

## Catching exceptions

Resource mangement: destructors and "stack unwinding"

```
struct Foo {
    int x;
    Foo(int ix) :x{ix} {
        cout << "Foo("<<x<<")\n";
    }
    ~Foo() {
        cout << "~Foo("<<x<<")\n";
    }
};

void test(int i)
{
    Foo f(i);
    if(i == 0) {
        throw std::out_of_range("noll?");
    } else {
        Foo g(100+i);
        test(i-1);
        cout << "after call to test("
            << i-1 << ")\n";
    }
}
```

```
int main() {
    Foo f(42);
    try{
        Foo g(17);
        test(2);
    } catch(std::exception& e) {
        cout<<e.what()<< endl; }
}
    Foo(42)
    Foo(17)
    Foo(2)
    Foo(102)
    Foo(1)
    Foo(101)
    Foo(0)
    ~Foo(0)
    ~Foo(101)
    ~Foo(1)
    ~Foo(102)
    ~Foo(2)
    ~Foo(17)
    ~Foo(42)
```

## Catching exceptions

Resource mangement: destructors and "stack unwinding"

```
struct Foo {
    int x;
    Foo(int ix) :x{ix} {
        cout << "Foo("<<x<<")\n";
    }
    ~Foo() {
        cout << "~Foo("<<x<<")\n";
    }
};

void test(int i)
{
    Foo f(i);
    if(i == 0) {
        throw std::out_of_range("noll?");
    } else {
        Foo g(100+i);
        test(i-1);
        cout << "after call to test("
            << i-1 << ")\n";
    }
}
```

```
int main() {
    Foo f(42);
    try{
        Foo g(17);
        test(2);
    } catch(std::exception& e) {
        cout<<e.what()<< endl; }
}
    Foo(42)
    Foo(17)
    Foo(2)
    Foo(102)
    Foo(1)
    Foo(101)
    Foo(0)
    ~Foo(0)
    ~Foo(101)
    ~Foo(1)
    ~Foo(102)
    ~Foo(2)
    ~Foo(17)
    ~Foo(42)
```

## Catching exceptions

Resource mangement: destructors and "stack unwinding"

```
struct Foo {
    int x;
    Foo(int ix) :x{ix} {
        cout << "Foo("<<x<<")\n";
    }
    ~Foo() {
        cout << "~Foo("<<x<<")\n";
    }
};

void test(int i)
{
    Foo f(i);
    if(i == 0) {
        throw std::out_of_range("noll?");
    } else {
        Foo g(100+i);
        test(i-1);
        cout << "after call to test("
            << i-1 << ")\n";
    }
}
```

```
int main() {
    Foo f(42);
    try{
        Foo g(17);
        test(2);
    } catch(std::exception& e) {
        cout<<e.what()<< endl; }
}
    Foo(42)
    Foo(17)
    Foo(2)
    Foo(102)
    Foo(1)
    Foo(101)
    Foo(0)
    ~Foo(0)
    ~Foo(101)
    ~Foo(1)
    ~Foo(102)
    ~Foo(2)
    ~Foo(17)
    ~Foo(42)
```

## Catching exceptions

Resource mangement: destructors and "stack unwinding"

```
struct Foo {
    int x;
    Foo(int ix) :x{ix} {
        cout << "Foo("<<x<<")\n";
    }
    ~Foo() {
        cout << "~Foo("<<x<<")\n";
    }
};

void test(int i)
{
    Foo f(i);
    if(i == 0) {
        throw std::out_of_range("noll?");
    } else {
        Foo g(100+i);
        test(i-1);
        cout << "after call to test("
            << i-1 << ")\n";
    }
}
```

```
int main() {
    Foo f(42);
    try{
        Foo g(17);
        test(2);
    } catch(std::exception& e) {
        cout<<e.what()<< endl; }
}
    Foo(42)
    Foo(17)
    Foo(2)
    Foo(102)
    Foo(1)
    Foo(101)
    Foo(0)
    ~Foo(0)
    ~Foo(101)
    ~Foo(1)
    ~Foo(102)
    ~Foo(2)
    ~Foo(17)
    ~Foo(42)
```

## Catching exceptions

Resource mangement: destructors and "stack unwinding"

```
struct Foo {
    int x;
    Foo(int ix) :x{ix} {
        cout << "Foo("<<x<<")\n";
    }
    ~Foo() {
        cout << "~Foo("<<x<<")\n";
    }
};

void test(int i)
{
    Foo f(i);
    if(i == 0) {
        throw std::out_of_range("noll?");
    } else {
        Foo g(100+i);
        test(i-1);
        cout << "after call to test("
            << i-1 << ")\n";
    }
}
```

```
int main() {
    Foo f(42);
    try{
        Foo g(17);
        test(2);
    } catch(std::exception& e) {
        cout<<e.what()<< endl; }
}
    Foo(42)
    Foo(17)
    Foo(2)
    Foo(102)
    Foo(1)
    Foo(101)
    Foo(0)
    ~Foo(0)
    ~Foo(101)
    ~Foo(1)
    ~Foo(102)
    ~Foo(2)
    ~Foo(17)
    ~Foo(42)
```

## Catching exceptions

Resource mangement: destructors and "stack unwinding"

```
struct Foo {
    int x;
    Foo(int ix) : x{ix} {
        cout << "Foo("<<x<<")\n";
    }
    ~Foo() {
        cout << "~Foo("<<x<<")\n";
    }
};

void test(int i)
{
    Foo f(i);
    if(i == 0) {
        throw std::out_of_range("noll?");
    }
    else {
        Foo g(100+i);
        test(i-1);
        cout << "after call to test("
            << i-1 << ")\n";
    }
}

int main() {
    Foo f(42);
    try{
        Foo g(17);
        test(2);
    } catch(std::exception& e) {
        cout<<e.what()<< endl; }
}

Foo(42)
Foo(17)
Foo(2)
Foo(102)
Foo(1)
Foo(101)
Foo(0)
~Foo(0)
~Foo(101)
~Foo(1)
~Foo(102)
~Foo(2)
~Foo(17)
noll?
~Foo(42)
```

## Specifying exceptions in C++11

The keyword `noexcept` specifies if a function may throw or not. No specification is equal to `noexcept(false)`.

In the function declaration

```
struct Foo {
    void f();
    void g() noexcept;
};
```

and in the function definition

```
#include<stdexcept>
void Foo::f() {
    throw std::runtime_error("f failed");
}
void Foo::g() noexcept{
    throw std::runtime_error("g lied and failed");
}
```

## Exception specification

Example usage

```
#include<typeinfo> // for typeid
void test_noexcept()
{
    Foo f;

    try {
        f.f();
    } catch(std::exception &e) {
        cout << typeid(e).name() << ": " << e.what() << endl;
    }
    try {
        f.g();
    } catch(std::exception &e) {
        cout << typeid(e).name() << ": " << e.what() << endl;
    }
    cout << "done\n";
}

St13runtime_error: f failed
terminate called after throwing an instance of 'std::runtime_error'
what(): g lied and failed
```

## Exception specification

older C++, do not use

Older C++ had "exception lists" for a function: the types of exceptions that may be generated by the function are specified with the keyword `throw`.

Example of exception list:

```
int f(int) throw(typ1, typ2, typ3) {
    //...
    throw typ1("Error of type 1 occurred");
    //...
    throw typ2("Error of type 2 occurred");
    //...
    throw typ3("Error of type 3 occurred");
}
```

No list ⇒ Any type of exception may be thrown  
Empty list (`throw()`) ⇒ No exceptions may be thrown

## Rules of thumb for exceptions

- ▶ Consider error handling early in the design process
- ▶ Use specific exception types, not built-in types. (do not `throw 17;`, `throw false;`, etc.)
- ▶ "Throw by value, catch by reference"
- ▶ If a function should not throw, declare `noexcept`.
- ▶ Specify *invariants* for your types
  - ▶ The constructor establishes the invariant, or throws.
  - ▶ Member functions can rely on the invariant.
  - ▶ Member functions must not break the invariant.
  - ▶ Example: Vector
    - ▶ the size `sz` is a positive number
    - ▶ the array `elem` points to has size `sz`
    - ▶ if the allocation of the array fails `std::bad_alloc` is thrown

If something can be checked at compile-time, use `static_assert`.

## Static assert

If something can be checked at compile-time, use `static_assert`.

```
constexpr int some_param = 10;

int foo(int x)
{
    static_assert(some_param > 100, "some_param too small");
    return 2*x;
}

int main()
{
    int x = foo(5);

    std::cout << "x is " << x << std::endl;
    return 0;
}

// error: static assertion failed: some_param too small
```

assert  
in <cassert>

A C standard macro for *fail fast* (at run-time).

```
void assert(some expression);
```

Prints an error message and calls `std::abort()` if the value of the expression is `false`.

### Example

```
#include <cassert>

std::vector<int> create_vector(int i)
{
    assert(i>=0);
    return std::vector<int>(i);
}
```

Typically only used during development.  
If `NDEBUG` is defined, it generates no code.

## Type casts

### Implicit conversions

#### Automatic conversions

- ▶ Expressions of the type  $x \odot y$ , for some binary operator  $\odot$   
E.g.: `double + int ==> double`  
`float + long + char ==> float`
- ▶ Assignments and initialization: The value of the right-hand-side is converted to the type of the left-hand-side
- ▶ Conversion of an argument to the type of the (formal) parameter
- ▶ Expressions in `if` statements, etc.  $\Rightarrow$  `bool`
- ▶ built-in array  $\Rightarrow$  pointer (*array decay*)
- ▶ `0`  $\Rightarrow$  `nullptr` (empty pointer in C++11, previously the constant `NULL` was defined)

## type casts

Explicit, named type casts (C++11)

- ▶ `static_cast<new_type> (expr)`
  - convert between compatible types (*does not do range check*)
  - "the inverse of a *standard* implicit conversion" (mostly)
- ▶ `reinterpret_cast<new_type> (expr)`
  - no safety net, same as C-style cast
- ▶ `const_cast<new_type> (expr)` - add or remove `const`
- ▶ `dynamic_cast<new_type> (expr)` - use for pointers to objects in class hierarchies. Uses *run-time type info*, like `instanceof` in Java.

### Example

```
char c; // 1 byte
int *p = (int*) &c; // pointer to int: 4 bytes

*p = 5; // undefined behaviour, stack corruption

int *q = static_cast<int*> (&c); // compiler error
```

## Type casting

Explicit type casts, C style

### Syntax in C and in C++, like in Java

(type) expression , e.g. `(float) 10`

- ▶ Greater risk of mistakes - use named casts
  - ▶ makes the code clearer, e.g., `const_cast` can only change `const`
  - ▶ easy to search for: casts are among the first to look for when debugging
- ▶ Warning in GCC: `-Wold-style-casts`
- ▶ Common in older code

### Alternative syntax in C++

type(expression)  
type must be a *single word*,  
`int *(...)` eller i.e., `unsigned long(...)` is not OK.

## Type casts

### Warning example

```
struct Point{
    int x;
    int y;
};

struct Point3d {
    int x;
    int y;
    int z;
};
```

Point: 

x:
y:

Point3d: 

x:
y:
z:

## Data types and variables

- ▶ some concepts:
  - ▶ a *type* defines the set of possible values and operations (for an *object*)
  - ▶ an *object* is a place in memory that holds a *value*
  - ▶ a *value* is a set of bits interpreted according to a *type*.

*A typecast changes the value of a particular memory location by changing how it should be interpreted.*

## Type casts

### Warning example

```

struct Point{
    int x;
    int y;
};
Point ps[3];

struct Point3d{
    int x;
    int y;
    int z;
};
Point3d* foo = (Point3d*) ps;
    
```

With *named casts*, this requires a `reinterpret_cast<Point3d*>`

With `static_cast<Point3d*>` the compiler gives the error `invalid static_cast from type 'Point[3]' to type 'Point3d*'`

## special case: void pointer

A `void*` can point to an object of any type

In **C** a `void*` is implicitly converted to/from any pointer type.

In **C++** a `T*` is implicitly converted to `void*`. The other direction requires an explicit *type cast*.

## Type casts

Example: `const_cast`: avoid code duplication

```

class X {
public:
    //...

    const Z& getZ(size_t index) const {return foo?bar:baz;}
    Z& getZ(size_t index) {return foo?bar:baz;}
};
    
```

*assumption*: the logic is the same. the only difference is the **constness** of the returned reference.

*problem*: duplicated code for `const` / non-`const`-version of `getZ()`.

*observation*: if an object of type `X` is *not const* it is safe to

- ① use `getZ(.) const` and
- ② remove `const` from the return value.

Rule: Call the `const` version from the non-`const` function.

## Type casts

Example: `const_cast`: avoid code duplication

```

class X {
public:
    //...

    const Z& getZ(size_t index) const {
        // really-really-really long access and checking code
    }

    Z& getZ(size_t index) {
        return const_cast<Z&>( const_cast<const X&>(*this).getZ(index));
    }

    #if 0
    // A slightly less-ugly version: two lines, one cast
    Z& getZ(size_t index) {
        const X& constMe = *this;
        return const_cast<Z&>( constMe.getZ(index) );
    }
    #endif
};
    
```

## Suggested reading

References to sections in Lippman  
[Error handling, exceptions](#) (5.6, 18.1.1)  
[static assert](#) *not in Lippman*  
[assert](#) 6.5.3

## Next lecture

References to sections in Lippman  
[Classes, copy control](#) 13.5 – 13.6  
[swap](#) 13.3  
[Container Adapters](#) 9.6  
[Pairs](#) 11.2.3  
[Tuples](#) 17.1