Tampereen yliopisto
Tampere University

Errors and exception handling

5.9.2019

Handling errors

- It can be difficult to anticipate errors and react to them:
  - Messy code
  - Need for cleaning
  - Recovery -> cancellation of unfinished tasks

Error?

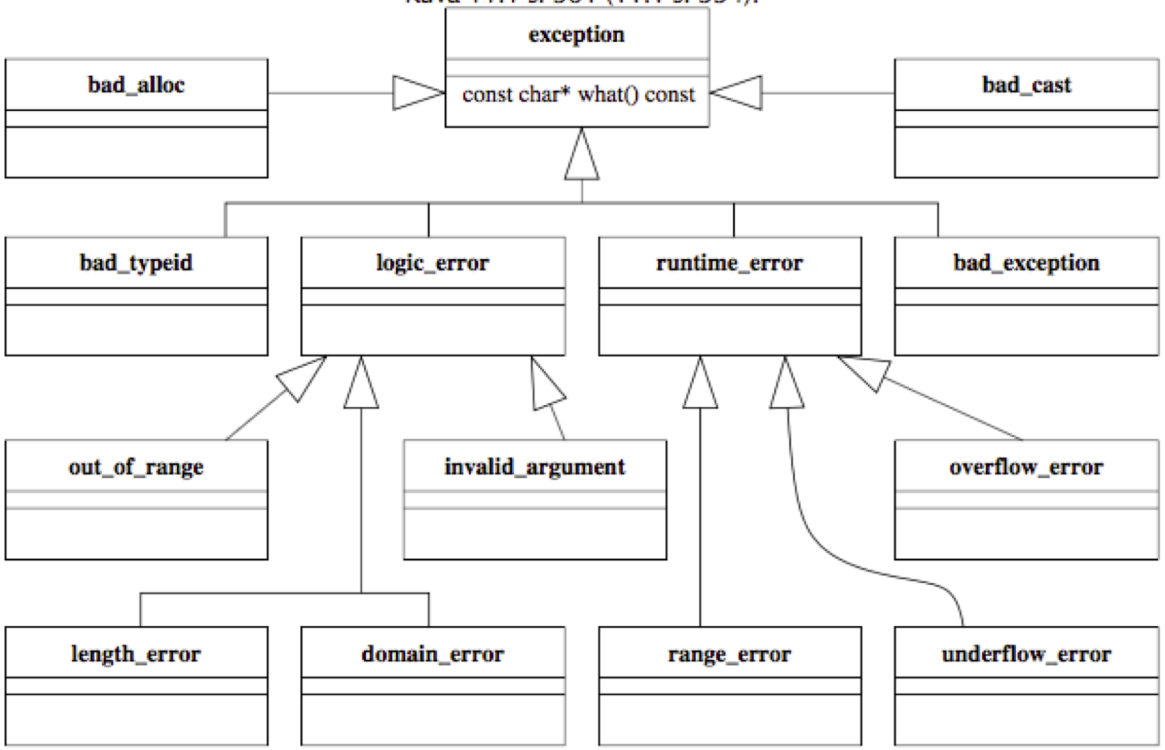
- External error: program is asked to do something, which it cannot do
- Internal error: execution leads to a situations, where something goes wrong
- Detecting erroneous situation is the easiest thing

Defensive programming

- Anticipating that other ones do something wrong
- What can be done when detecting an error?
  - Abrupt termination
  - Controlled termination (abort, exit)
  - Continuation
  - Cancellation (rollback)
  - Recovery

Error hierarchies

Kuva 11.1 s. 301 (11.1 s. 334):



363 *Olioiden ohjelmointi C++:lla,* Matti Rintala & Jyke Jokinen1941

Tampereen yliopisto
Tampere University

Errors as C++ classes

```
// In std namespace
class exception {
public:
    exception() throw(); // throw() will be explained later
    exception(exception const& e) throw();
    exception& operator =(exception const& e) throw();
    virtual ~exception() throw();
    virtual char const* what() const throw();
    ...
};

class runtime_error : public exception {
public:
    runtime_error(std::string const& msg);
};
class overflow_error : public runtime_error {
public:
    overflow_error(std::string const& msg);
};
```

Using exception hierarchy

```
class ValueTooSmall : public std::domain_error
{
public:
    ValueTooSmall(std::string const& message,
                  int number, int minimum);
    ValueTooSmall(ValueTooSmall const& error);
    virtual ~ValueTooSmall() throw();
    int giveNumber(() const;
    int giveMinimum() const;
private:
    int number_;
    int minimum_;
};
```

Exception handling

- Code prone to an error is written in **try** block
- When detecting an error, an exception is **throw**n
- Exception handler **catch**es the exception and processes the error
- After handling an error:
  - No return to error occurrence, but execution continues after the error handling structure

Exception handling

- Looking for an exception handler
  - Backtracking in the call chain until a suitable handler is found
- Parameter instructs:
  - Which types of errors can be caught
  - Reference to the base class of an error -> any error of a subclass
- Or: no parameter

Tampereen yliopisto
Tampere University

Example

```
void readNumbersToTable(vector<double>& table);

double sumNumbers(vector<double> const& numbers) {
    for (unsigned int i = 0; i < numbers.size(); ++i) {
        if (sum >= 0 && numbers[i] > numeric_limits<double>::max() - sum)
        {
            throw std::overflow_error("Sum is too  big");
        }
        else if (sum < 0 && numbers[i] < -numeric_limits<double>::max() - sum)
        {
            throw std::overflow_error("Sum is too small");
        }
        sum += numbers[i];
    }
    return sum;
}
```

Example (cont.)

```
double calculateAverage (vector<double> const& numbers) {
    unsigned int count = numbers.size();
    if (count == 0)
    {
        throw std::range_error(“Count in average is 0");
    }
    return sumNumbers(numbers) / static_cast<double>(count);
}
```

Tampereen yliopisto
Tampere University

Example (cont.)

```
void average1(vector<double>& numberTable) {
    try
    {
        readNumbersToTable(numberTable);
        double average = calculateAverage(numberTable);
        cout << “Average: " << average << endl;
    }
    catch (std::range_error const& error)
    {
         cerr << “Range error: " << error.what() << endl;
    }
    catch (std::overflow_error const& error)
    {
        cerr << “Overflow: " << error.what() << endl;
    }
    cout << “End" << endl;
}
```

Tampereen yliopisto
Tampere University

Error categories

```
void Average2(vector<double>& numberTable) {
    try
    {
        readNumbersToTable(numberTable);
        double average = calculateAverage (numberTable);
        cout << “Average: " << average << endl;
    }
    catch (std::runtime_error const& error)
    {
        // Any run-time error leads to here
        cerr << “Run-time error: " << error.what() << endl;
    }
    cout << “End" << endl;
}
```

Exceptions not caught

- E.g. memory exhaustion
  - Calling **terminate** function
    - Terminates the program (and gives an error message)
  - Programmer can write an own implementation for terminate function

Catch-all handler

- Catches any exception
  - Promises to handle all errors! -> do not use in vain!
  - Useful in cleaning, if an exception is thrown again

```
catch (...) // Really ... Three dots
{
    // This handler catches any exception
}
```

Tampereen yliopisto
Tampere University

Nested try blocks

```
int main() {
    vector<double> table;
    try {
        average2(table); // Read numbers and calculate their average
    }
    catch (std::bad_alloc&) {
        cerr << "Memory exhaustion!" << endl;
        return EXIT_FAILURE;
    }
    catch (std::exception const& error) {
        cerr << “Error in main program: " << error.what() << endl;
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Exceptions & deletion of objects

- Looking for an exception handler -> exit from a code block -> lifetime of an object may end
- -> such objects are deleted automatically
- -> exceptions do not cause problems, if the lifetime of an object is static

Exceptions & destructors

- Single exception per a block
  - Destructor must not leak the exception!
  - If this happens -> terminate
  - Usually no problems (cleaning succeeds)

Exceptions & dynamic objects

- Objects created dynamically do not have lifetime controlled by a program structure
- -> Exception handling mechanism do not automatically delete objects created by **new** command
- Local pointers are deleted -> dangling pointers

Exceptions & dynamic objects

- Solution 1:
  - Catch all errors, **delete** objects, throw the exception again
  - Be careful, if there are several objects (error may have occurred before creating all of them)
- Solution 2: smart pointers

Tampereen yliopisto
Tampere University

Solution 1

```
void cleanFunction()
{
    vector<double>* tablep = new vector<double>();
    try
    {   // If an error occurs here, the vector must be deleted
        average2(*tablep);
    }
    catch (…)
    {   // Catching all errors and deleting the vector
        delete tablep; tablep = 0;
        throw;  // Throws the exception to further handling
    }
    delete tablep; tablep = 0;  // This is reached, if no errors
                                // occur
}
```

Exception safety

- Encapsulation hides information about the implementation — also about possible errors
- Inheritance & polymorphism — implementation even more far away and it can vary
- **-> Interface documentation** highly important

Exception safety

- Subclasses must not violate the promises given their base classes
- Base class must not promise too much about the errors or the lack of them
- Documentation becomes easier by using predefined terms for different situations: **exception guarantees**

Exception guarantees

- **Minimal guarantee** – No waste of resources
  - object can be deleted/reset but not otherwise usable
  - class invariant do not necessarily hold

Exception guarantees

- **Basic guarantee**
  - state of an object non-predictable but valid
  - class invariant continues to hold
  - object still usable per se

Exception guarantees

- **Strong guarantee**
  - Commit or rollback
- **Nothrow guarantee**
  - No errors happen, ideal for the programmer
- **Exception neutrality**