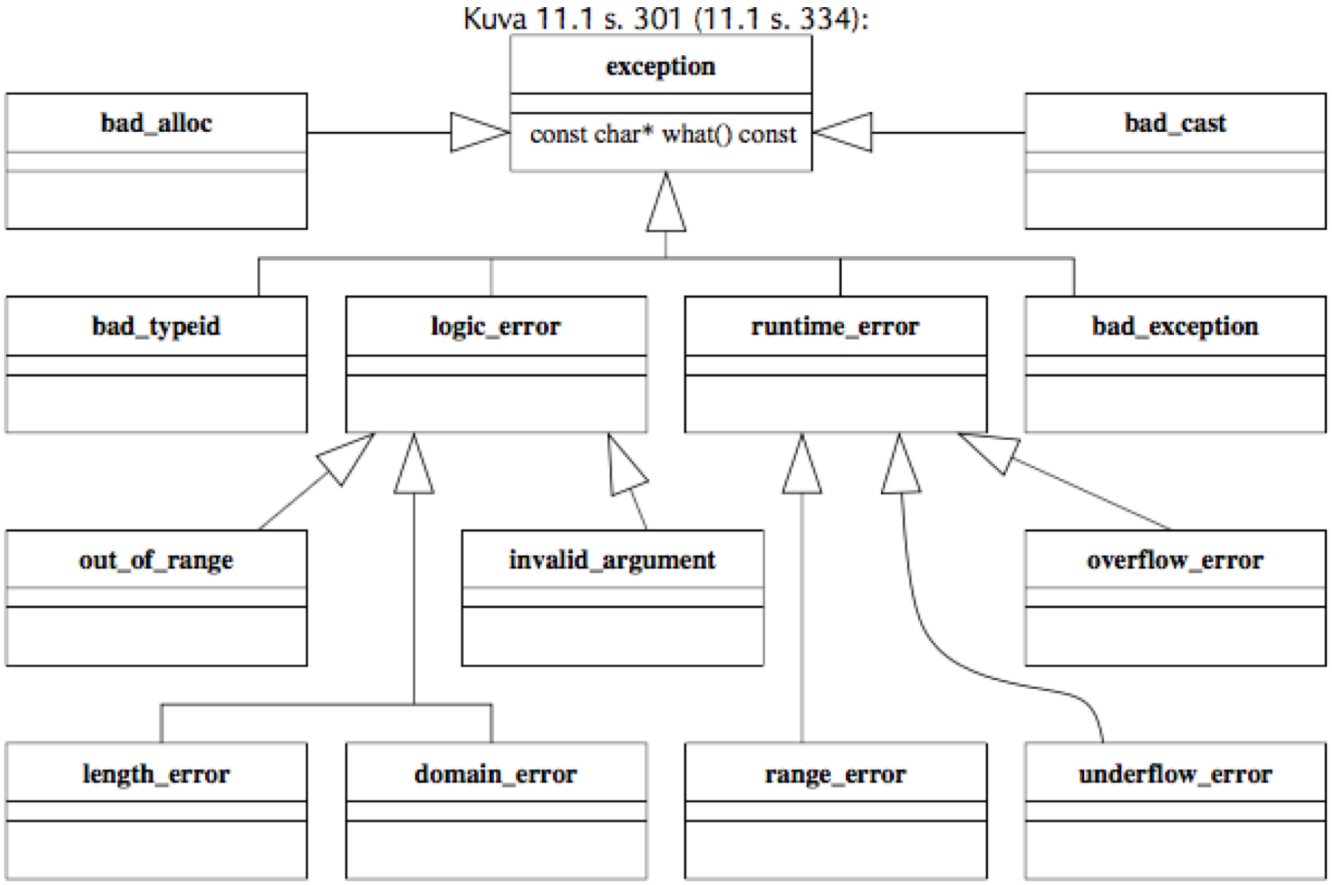
Tampereen yliopisto
Tampere University

Exception safety

2.10.2018

Tampereen yliopisto
Tampere University

Error hierarchy



Exceptions and constructors

- Objects are constructed step by step — when is the time of creation?
- Important to know, if construction leads to errors
- C++: object “is born” when all the constructors have been executed successfully

Exceptions and constructors

- A single error unhandled in the constructor → the object does not exist
- Destructors of those member variables that have already been constructed will be executed
- Be careful, if constructors have dynamically created objects

Exceptions and constructors

```
class Person {
public:
    Person(int d, int m, int y, std::string const& name,
           std::string const& id);
    ~Person();
private:
    Date birthDay_;
    std::string name_;
    std::string* id_;
};
```

Exceptions and constructors

```
Person::Person(int d, int m, int y, std::string const& name,
                        std::string const& id)
  : birthDay_(d, m, y), name_(name), id_(0) {
    try {
        id_ = new std::string(id);
    }
    catch (...) {
        // If here, then creation of identification failed
        // Clean up if necessary, creation of the object failed
        throw; // throw the exception to be handled further
    }
}
```

Reaction to creation errors

- Constructor of a member variable or that of the base class fails → creation cannot succeed
- Dynamically created object can try to be recreated, if reasonable
- Errors in the constructors of member variables and that of the base class caught in *function try block*

Reacting to creation errors

- Error can be changed to another one, recovery not possible
- Member variables and base class parts have already been destructed → they cannot be accessed

Function try block in constructor

```
Person::Person(int d, int m, int y, std::string const& name,
               std::string const& id)
try // Note the place of try!
  : birthDay_(d, m, y), name_(name), id_(0) {
    …
}
catch (...) { // If here, then creation of a member variable
              // (or base class) has failed.
    // Necessary actions.
    throw; // … or another exception is thrown.
}
```

Exceptions and destructors

- Constructors should not leak out exceptions!
- Constructor should handle the exceptions caused by itself
- If not, then it is better to use a special member function to clean up
- Function try block is possible in principle, but almost as useless in destructors
- `uncaught_exception` — also almost useless

Exception safety

- Encapsulation hides implementation — as well as the risks of errors
- Inheritance & polymorphism — implementation even more far away and varying

→ **interface documentation** extremely important

Exception guarantees

- Subclasses must not violate promises given in their base classes
- Base class must not promise too much about errors or the lack of them
- Documentation becomes easier by 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**

Example: Exception safe assignment

- Recall for class Book: analysis and improvement of the assignment operator
- **Step 1**: analyze existing exception guarantee
- **Step 2**: improve it, if possible and rational

Simple class with assignment operator

```
class Book {
public:
    …
    Book& operator =(Book
                const& book);
private:
    std::string title_;
    std::string author_;
};
```

```
Book& Book::operator =(Book const&
book)
{
    if (this != &book) {
        title_ = book.title_;
        author_ = book.author_;
    }
    return *this;
}
```

```
Book& Book::operator =(Book const& book) {
    if (this != &book) {
        std::string origTitle(title_);
        std::string origAuthor(author_);
        try {
            title_ = book.title_;
            author_ = book.author_;
        }
        catch (...) {
            title_ = origTitle;
            author_ = origAuthor;
            throw;
        }
    }
    return *this;
}
```

Goal: strong guarantee

More indirect solution

```
class Book {
public:
    Book(std::string const& title, std::string const& author);
    // Copy constructor also needed (dynamic memory management)!
    ~Book();
    // ...
    Book& operator =(Book const& book);
private:
    std::string* titlep_;
    std::string* authorp_;
};
```

Strong guarantee in indirect case

```
Book::Book(std::string const& title, std::string const& author) :
            titlep_(0), authorp_(0) {
    try {
        titlep_ = new std::string(title);
        authorp_ = new std::string(author);
    }
    catch (...) {
        delete titlep_; titlep_ = 0;
        delete authorp_; authorp_ = 0;
        throw;
    }
}
```

Similarly:

```
Book::~Book() {
    delete titlep_; titlep_ = 0;
    delete authorp_; authorp_ = 0;
}
```

```
Book& Book::operator =(Book const& book) {
    if (this != &book) /* Actually unnecessary! */ {
        std::string* newTitlep = 0;
        std::string* newAuthorp = 0;
        try {
            newTitlep = new std::string(*book.titlep_);
            newAuthorp = new std::string(*book.authorp_);
            // If here, then no errors detected
            delete titlep_; titlep_ = newTitlep; // Succeed always
            delete authorp_; authorp_ = newAuthorp; // As above
        }
        catch (...) {
            delete newTitlep; newTitlep = 0;
            delete newAuthorp; newAuthorp = 0;
            throw;
        }
    }
    return *this;
}
```

… cont.

Private implementation (*pimpl*)

```
class Book {
public:
    Book(std::string const& title, std::string const& author);
    // Copy constructor also needed!
    ~Book();
    // ...
    Book& operator =(Book const& book);
private:
    struct State;
    std::unique_ptr<State> statep_;
};
```

Private implementation (*pimpl*)

```
struct Book::State {
    std::string title_;
    std::string author_;
    State(std::string const& title, std::string const& author) :
          title_(title), author_(author) {}
};

Book::Book(std::string const& title, std::string const& author) :
          statep_(new State(title, author))
{
}

Book::~Book()
{ // Unique pointer destructs the state automatically
}
```

Private implementation (*pimpl*)

```
Book& Book::operator =(Book const& book)
{
    std::unique_ptr<State> newStatep =
                        std::make_shared<State>(*book.statep_);
    statep_ = std::move(newStatep);  // Cannot fail and
                                     // destructs the old state
    return *this;
}
```

Swapping states (nothrow)

```
class Book
{
public:
    // ...
    Book& operator =(Book const& book);
    void swap(Book& book);
private:
    std::string title_;
    std::string author_;
};
```

Swapping states (nothrow)

```
void Book::swap(Book& book) {
    title_.swap(book.title_); // This cannot fail
    author_.swap(book.author_); // This neither
}

Book& Book::operator =(Book const& book) {
    Book bookCopy(book); // Copy of the book to be assigned
    swap(bookCopy); // Swapping ourselves to it, does not fail
    return *this; // Old state is destroyed along with the copy
}
```

C++ specifiers

- `override` (for virtual functions) → subclass provides an implementation of its own
- `final` (for virtual functions) → subclass cannot provide an implementation of its own
- **`noexcept`** → `no exception is thrown`
- `= 0` → pure virtual function