OOP (C++): Exceptions

Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iaşi, Romania
dlucanu@info.uaic.ro

Object Oriented Programming 2023/2024

Plan

What the Experts Say

- "... I realized that from now on a large part of my life would be spent finding and correcting my own mistakes."
  Maurice Wilkes, 1949 (EDSAC computer), Turing Award in 1967
- "My guess is that avoiding, finding, and correcting errors is 95% or more of the effort for serious software development."
  "Error handling is a difficult task for which the programmer needs all the help that can be provided."
  "When we write programs, errors are natural and unavoidable; the question is, how do we deal with them?"
  Bjarne Stroustrup (C++ creator), 2015[1]

This rate can be improved by using the right programming skills!

[1] https://www.stroustrup.com/PPP2slides/5_errors.ppt

Three Main Requirements for a Program

- to produce the desired outputs, as specified, for legal inputs (correctness)
- to give reasonable error messages for illegal entries
- to allow termination in case of an error

A Taxonomy of Errors

- compilation errors
  - detected by the compiler
  - generally easy to fix
- linking errors
  - functions/methods not implemented
  - library access
- run-time errors
  - the source may be the computer, a component of a library, or the program itself
- logical errors
  - the program does not have the desired behavior
  - can be detected by the programmer (by testing) . . . or by the user (customer)

Errors Must be Reported

```
int PolygonalLine::length() {
  if (n <= 0)
    return -1
  else {
    // ...
  }
}

int p = L.length();
if (p < 0)
  printError("Bad computation of a polygonal line");
// ...
```

How to Report an Error?

- error reporting must be uniform: same message for the same type of error (location information may differ)
- ...therefore an "error indicator" must be managed
- association of numbers for errors is a solution but it can be problematic
- OOP has the necessary means to smartly organize the detection and reporting of errors

Error vs Exception

- often the two terms are confused
- however, there is a (subtle) difference between the meanings of the two notions
- error = abnormal behavior detected during operation to be eliminated by repairing the program
- exception = unforeseen behavior that can occur in rare or very rare situations
- exceptions should be handled in the case of programs
- an untreated exception is an error!

In this lecture we discuss more about exceptions.

Plan

General Principles

- the exceptions in OOP were designed with the intention of separating the business logic from the mechanism of error transmission

- the purpose is to allow the handling of errors, which occur as exceptions, at an appropriate level that does not interfere with business logic

Plan

Syntax for Regular Expressions

We use Backus-Naur Form (BNF) notation to specify the syntax:

```
expression ::= term ("+" term)*
term ::= maybeStar ("." maybeStar)*
maybeStar ::= factor ["*"]
           /* equiv to  factor | factor "*" */
factor ::=
     empty
     | Sigma
     | "(" expression ")"
 Sigma ::= "a" | "b" | ...
```

Syntax for Regular Expressions Explained

```
expression ::= term ("+" term)*
```

An `expression` is a term followed by a possible empty sequence of pairs `+ term`.

```
maybeStar ::= factor ["*"]
              /* equiv to  factor | factor "*" */
```

A `maybeStar` is a `factor` optionally followed by a `*`.

```
factor ::=
    empty
    | Sigma
    | "(" expression ")"
```

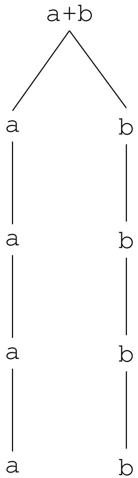
A `factor` is either `empty`, or a `Sigma`, or an `expression` included in brackets.

```
Sigma ::= "a" | "b" | ...
```

A `Sigma` (alphabet) is either `a` or `b` or . . . .

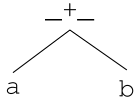
Example of Regular Expression and Derivation

```
expression ::= term "+" term
term ::= maybeStar
maybeStar ::= factor
factor ::= Sigma
Sigma ::= "a"
Sigma ::= "b"
```



Abstract Syntax Tree (AST) Example 1/2

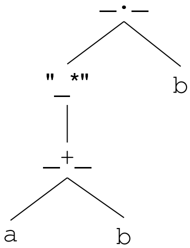
```
a + b
```




```
["_+_", < ["a", <>], ["b", <>]>]
```

Abstract Syntax Tree (AST) Example 2/2

(a + b)* b




```
["_._", <["_*", <["_+_", <["a", <>],
                           ["b", <>]>]>],
        ["b", <>], >]
```

Abstract Syntax Tree (AST) Definition

$ast(e)$

- *empty*: `[]`
- ε: `["", <>]`
- $a \in \Sigma$: `["a", <>]`
- $e_1 e_2 \ldots$: $\texttt{["\_.\_",}\ < ast(e_1), ast(e_2), \texttt{...} > \texttt{]}$
- $e_1 + e_2 + \cdots$: $\texttt{["\_+\_",}\ < ast(e_1), ast(e_2), \cdots > \texttt{]}$
- $e*$: $\texttt{["\_*",}\ < ast(e) > \texttt{]}$

Some functions from AST domain

```
// number of children
chldNo(ast) {
  if (ast.size() > 0) return ast[1].size();
  return 0;
}

// the i-th child of an AST
chld(ast, i) {
  if (ast.size() > 0 && i < ast[1].size()) {
    return ast[1].at(i);
  }
}
```

(Alk notation)

Plan

Class for ASTs, First Attempt

```
class Ast {
 private:
  string label;
  vector<Ast*> children;

 public:
  AstNonEmpty(string aLabel = "",
              vector<Ast*> aList = vector<Ast*>())
    {
      label = aLabel;
      children = aList;
    }
  ...
};
```

We have a problem: how to represent the empty AST ("[]")?

In the above declaration, the default constructor builds the AST for the empty string ε.

Class for ASTs, Second Attempt

```
class Ast {        //abstract class
 public:
  virtual int chldNo() = 0;
  virtual Ast* child(int i) = 0;
  ...
};
class AstEmpty : public Ast {       //empty AST []
 public:
  AstEmpty() {}       //create an object representing the empty AST
  ...
};
class AstNonEmpty : public Ast {    //non empty AST ["...",<...>]
 private:
  string label;
  vector<Ast*> children;
 public:
  AstNonEmpty(string aLabel = "",
              vector<Ast*> aList = vector<Ast*>())
    {
      label = aLabel;
      children = aList;
    }
  ...
};
```

Implementation of `AstEmpty` (partial)

```
int AstEmpty::chldNo() {
   #??   is it OK to return -1? ⇐ exception
}
Ast* AstEmpty::child(int i) {
   #??   is it OK to return new AstEmpty? ⇐ exception
}
```

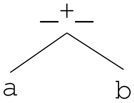
Implementation of `AstNonEmpty` (partial)

```
Ast* AstNonEmpty::child(int i) {
  if (0 <= i and i < chldNo()) {
    return children[i];
  }
  #??   is it OK to return new AstEmpty? ⇐ exception
}
```

Parsing Regular Expressions

Parser: Regular Expression e as a String $\rightarrow$ AST(e)

`"a + b"` $\mapsto$



The Parser

Global variables:

`input` - the expression given as input

`sigma` - the alphabet

`index` - the current position in the input

- the current symbol:

```
sym() modifies index uses input {
  if (index < input.size())
    return input.at(index);
  return "\0";
}
```

- next symbol

```
nextSym() modifies index uses input {
  if (index < input.size()) {
    index++;
  } else
    error("nextsym: expected a symbol");
}
```

- ...

(Alk notation)

The Parser in C++ 1/3

```
class Parser {
 private:
  string input;
  vector<char> sigma;
  int index;
 public:
  Parser(vector<char> alphabet = vector<char>()) {
    sigma = alphabet;
    input = "";
    index = 0;
  }
 ....
```

The Parser in C++ 2/3

```
char Parser::sym() {
  if (index < input.size()) {
    return input[index];
  }
  return '\0';
}

void Parser::nextSym() {
  if (index < input.size()) {
    index++;
  }
  else {
    error("nextsym: expected a symbol"); // ⇐exception
  }
}
```

The Parser in C++ 3/3

```
factor ::=
    empty
    | Sigma
    | "(" expression ")"
```

Alk	C++

```
factor() {
    s = sym();
    if (acceptSigma()) {
      return [s,<>];
    } else if (accept("(")) {
        ast = expression();
        expect(")");
        return ast;
    }
}
```

```
Ast∗ Parser::factor() {
  Ast∗ past;
  char s = sym();
  if (acceptSigma()) {
    past =
      new AstNonEmpty(string(1, s));
  } else if (accept('(')) {
    past = expression();
    expect(')');
  }
  // else ?? ⇐ exception
  return past;
}
```

Similar the other methods.

Testing

Test source:

```
int main() {
  vector<char> alph{'a', 'b', 'c'};  // c++11
  Parser parser(alph);
  parser.setInput("(a.b+");  // wrong expression
  past = parser.expression();
  past->print();
  return 0;
}
```

Test Run:

```
$ g++ ast.cpp parser.cpp test-parser.cpp -std=c++11
$ ./a.out
Bus error: 10
```

Plan

try, throw and catch

From the manual:

throw expression signals that an exceptional condition—often, an error—has occurred in a try block. You can use an object of any type as the operand of a throw expression. Typically, this object is used to communicate information about the error

try block is used to enclose one or more statements that might throw an exception (business component)

catch blocks are implemented immediately following a try block. Each catch block specifies the type of exception it can handle (error handling component)

`Ast` Hierarchy with Exceptions

Use `throw` whenever an exceptions occurs in methods!

Consider only problematic methods:

```
int AstEmpty::chldNo() {
  throw "Error: trying to access children of an empty AST";
}

Ast* AstEmpty::child(int i) {
  throw "Error: trying to access children of an empty AST";
}

Ast* AstNonEmpty::child(int i) {
if (0 <= i and i < chldNo()) {
  return children[i];
}
throw "Error: index out of bounds.";
}
```

Is it enough?

How a code including `throw` is executed?

Since it may throw exceptions, we can ONLY `try` to execute it.

What happens when a tried code throw exceptions?

They must be `caught`, using `catch` , and treated.

Testing, w/o Try

Test source:

```
int main() {
  AstEmpty  ast;
  cout << "ast.child(1)->print(): ";
  ast.child(1)->print();
  cout << endl;
  cout << "ast.chldNo(): ";
  cout << ast.chldNo();
  cout << endl;
  return 0;
}
```

Test Run:

```
$ g++ ast.cpp parser.cpp test-ast-empty.cpp -std=c++11
$ ./a.out
libc++abi.dylib: terminating with uncaught exception
    of type char const*
ast.child(1)->print(): Abort trap: 6
```

Testing, w/ Try and Catch

Test source:

```
try {      //business logic block
  AstEmpty  ast;
  cout << "ast.child(1)->print(): ";
  ast.child(1)->print();
  cout << endl;
  cout << "ast.chldNo(): ";
  cout << ast.chldNo();
  cout << endl;
}
catch (char const* msg) { //exception handling block
  cout << msg << endl;
  cout << "Here the exceptions can be handled." << endl;
}
```

Test Run:

```
$ g++ ast.cpp parser.cpp test-ast-empty-w-try.cpp -std=c++11
$ ./a.out
ast.child(1)->print(): Error: trying to access children
     of an empty AST
Here the exceptions can be handled.
```

`Parser` Class with exceptions 1/2

Only problematic methods:

```
void Parser::nextSym() {
  if (index < input.size()) {
    index++;
  }
  else {
    throw "Error: index out of input size";
  }
}

//bool Parser::expect(char s) {
void Parser::expect(char s) {
  if (accept(s)) {
    // return true;
    return;
  }
  // error("unexpected symbol at position " + to_string(index)
  // return false;
  throw s;
}
```

Parser Class with exceptions 2/2

Only problematic methods:

```
Ast* Parser::factor() {
  Ast* past;
  char s = sym();
  if (acceptSigma()) {
    past = new AstNonEmpty(string(1, s));
  } else if (accept('(')) {
    past = expression();
    expect(')');
  }
  // else ???
  else
    throw "expected alphabet symbol or (.";
  return past;
}
```

Testing, w/ Try 1/3

Test source:

```
try{
  vector<char> alph{'a', 'b', 'c'};   // c++11
  Parser parser(alph);
  parser.setInput("(a.b+");   // wrong expression
  Ast* past = parser.expression();
  past->print();
}
catch (char const* msg) {
  cout << msg << endl;
  cout << "Here the parsing exceptions can be handled." << e
}
```

Test Run:

```
$ g++ ast.cpp parser.cpp test-parser-exc2.cpp -std=c++11
$ ./a.out
expected alphabet symbol or (.
Here the parsing exceptions can be handled.
```

Testing, w/ Try 2/3

Test source:

```
try{
  vector<char> alph{'a', 'b', 'c'};  // c++11
  Parser parser(alph);
  parser.setInput("(a.b+c");  // wrong expression
  Ast* past = parser.expression();
  past->print();
}
catch (char const* msg) {
  cout << msg << endl;
  cout << "Here the parsing exceptions can be handled." << e
}
```

Test Run:

```
$ g++ ast.cpp parser.cpp test-parser-exc2.cpp -std=c++11
$ ./a.out
libc++abi.dylib: terminating with uncaught exception
     of type char
Abort trap: 6
```

Testing, w/ Try 3/3

Test source:

```
try{
  vector<char> alph{'a', 'b', 'c'};   // c++11
  Parser parser(alph);
  parser.setInput("(a.b+c");   // wrong expression
  Ast* past = parser.expression();
  past->print();
}
catch (char const* msg) {   // catching C string exceptions
  cout << msg << endl;
  cout << "Here the parsing exceptions can be handled." << e
}
catch (char s) {   // catching char exceptions
  cout << "expected character " << s << endl;
  cout << "Here the parsing exceptions can be handled." << e
}
```

Test Run:

```
$ g++ ast.cpp parser.cpp test-parser-exc2.cpp -std=c++11
$ ./a.out
expected character )
Here the parsing exceptions can be handled.
```

Plan

Hierarchy of Exceptions 1/3

It is strongly recommended to organize the exceptions into a hierarchy.
AST exceptions I:

```
class MyException {
 public:
  virtual void debugPrint() {
    std::cout << "Exception: ";
  }
};

class AstException : public MyException {
 public:
  virtual void debugPrint() {
    this->MyException::debugPrint();
    std::cout << "AST: ";
  }
};

class EmptyAstException : public AstException {
 public:
  virtual void debugPrint() {
    this->AstException::debugPrint();
    std::cout << "trying to access an empty tree.";
  }
};
```

Hierarchy of Exceptions 2/3

AST exceptions II:

```
class NonEmptyAstException : public AstException {
 private:
  int badIndex;
 public:
  void setBadIndex(int j) {
    badIndex = j;
  }
  virtual void debugPrint() {
    this->AstException::debugPrint();
    std::cout << "index out of bounds (" << badIndex << ").\n"
  }
};
```

Hierarchy of Exceptions 3/3

Parser exceptions:

```
class ParserException : public MyException {
 public:
 private:
  int badPos;
  std::string errMsg;
 public:
  void setBadPos(int j) {
    badPos = j;
  }
  void setErrMsg(std::string msg) {
    errMsg = msg;
  }
  virtual void debugPrint() {
    this->MyException::debugPrint();
    std::cout << "Parsing: " << errMsg
              << " at input position " << badPos << ".\n";
  }
};
```

Ast Hierarchy with exceptions, revisited

Consider only problematic methods:

```
int AstEmpty::chldNo() {
  throw EmptyAstException();
}

Ast* AstEmpty::child(int i) {
   throw EmptyAstException();
}

Ast* AstNonEmpty::child(int i) {
   if (0 <= i and i < chldNo()) {
     return children[i];
   }
   NonEmptyAstException exc;
   exc.setBadIndex(i);
   throw exc;
}
```

`Parser` Class with exceptions, revisited 1/2

Only problematic methods:

```
void Parser::nextSym() {
  if (index < input.size()) {
    index++;
  }
  else {
    ParserException exc;
    exc.setBadPos(index);
    throw exc;
  }
}

void Parser::expect(char s) {
  if (accept(s)) {
    return;
  }
  ParserException exc;
  exc.setErrMsg(string("unexpected symbol"));
  exc.setBadPos(index);
  throw exc;
}
```

Parser Class with exceptions, revisited 2/2

Only problematic methods:

```
Ast* Parser::factor() {
  Ast* past;
  char s = sym();
  if (acceptSigma()) {
    past = new AstNonEmpty(string(1, s));
  } else if (accept('(')) {
    past = expression();
    expect(')');
  } // else ???
  else {
    ParserException exc;
    exc.setErrMsg(string("expected alphabet symbol or ("));
    exc.setBadPos(index);
    throw exc;
  }
  return past;
}
```

Testing 1/3

Menu function:

```
void printMenu() {
  cout << "\n1. Empty AST exception.\n";
  cout << "2. Nonempty AST exception.\n";
  cout << "3. Parser exception.\n";
  cout << "0. Exit.\n";
  cout << "Option: ";
}
```

Testing 2/3

Test source:

```
while (opt != 0) {
  try{
    printMenu();
    cin >> opt;
    switch (opt) {
    case 1: {
      AstEmpty* peast = new AstEmpty();
      peast->child(0)->print();
      break;
    }
    ...
    }
  }
  catch (MyException& exc) {
    exc.debugPrint();
    cout « endl;
    parser.setIndex(0);
  }
}
```

Testing 3/3

Test Run:

```
$ ./a.out
1. Empty AST exception.
2. Nonempty AST exception.
3. Parser exception.
0. Exit.
Option: 1
Exception:  AST: trying to access an empty tree.
1. Empty AST exception.
2. Nonempty AST exception.
3. Parser exception.
0. Exit.
Option: 2
Exception:  AST: index out of bounds (2).
1. Empty AST exception.
2. Nonempty AST exception.
3. Parser exception.
0. Exit.
Option: 3
Exception:  Parsing:  expected alphabet symbol or (at input position
7.
1. Empty AST exception.
2. Nonempty AST exception.
3. Parser exception.
0. Exit.
Option: 0
```

Plan

Dynamic Exception Specification: Example (until C++ 2011)

```
Ast* Parser::factor() throw(ParserException) {
  Ast* past;
  char s = sym();
  if (acceptSigma()) {
    past = new AstNonEmpty(string(1, s));
  } else if (accept('(')) {
    past = expression();
    expect(')');
  } else {
    ParserException* pexc = new ParserException();
    pexc->setErrMsg(string("expected alphabet symbol or ("));
    pexc->setBadPos(index);
    throw pexc;
  }
  return past;
}
```

Dynamic Exception Specification

`throw ( type-id-list (optional))`

Depricated since C++2011.

Removed in C++17

Reason[2]:
"The recommendation of this paper is to remove dynamic exception specifications from the language. However, the syntax of the throw() specification should be retained, but no longer as a dynamic exception specification. Its meaning should become strictly an alias for noexcept(true), and its usage should remain deprecated.
...
Dynamic exception specifications are a failed experiment, but this is not immediately clear to novices, ...
Define semantics consistently in the terms of potentially-throwing or non-throwing operations, rather than whether exceptions are allowed ..."

[2] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0003r4.html

Current Exception Specification

- Since C++ 2011:

  ```
  noexcept(boolean-xpression);
  ```

Example 1/3

Exceptions in destructors may lead to unpredictable behavior, therefore they should not throw exceptions by default:

```
void f1()
{
    throw "Exception f1";
}

void f2() noexcept(false)
{
    throw "Exception f2";
}

void g() noexcept  // equivalent to "noexcept(true)"
{
    cout << "Execute g\n";
}
```

Example 2/3

Output:

```
1. f1.
2. f2.
3. g.
Option: 1
Exception f1

1. f1.
2. f2.
3. g.
Option: 2
Exception f2

1. f1.
2. f2.
3. g.
Option: 3
Execute g

1. f1.
2. f2.
3. g.
Option: 0
```

Example 3/3

Exceptions in destructors may lead to unpredictable behavior, therefore they should not throw exceptions by default:

```
void g() noexcept  // equivalent to "noexcept(true)""
{
    throw "Execute g\n";
}

int main(void)
{
  try {
      g();
  } catch(const char * msg) {
      cout << msg << '\n';
  }
}
```

Output:

```
misc/noexcept-clause-err.cpp:8:5: warning: 'g' has a non-throw
    throw "Execute g";
    ^
misc/noexcept-clause-err.cpp:6:6: note: function declared non-
void g() noexcept  // equivalent to "noexcept(true)""
     ^   ~~~~~~~~
1 warning generated.
```

Exception in Destructor 1/3

Exceptions in destructors may lead to unpredictable behavior, therefore they should not throw exceptions by default:

```
class A {
    public:
        ~A() {
            throw "Thrown by Destructor";
        }
};
int main() {
    try {
        A   a;
    }
    catch(const char *exc){
      std::cout << "Print " << exc;
    }
```

Output:

```
libc++abi.dylib: terminating with uncaught exception
    of type char const*
Abort trap: 6
```

Exception in Destructor 2/3

However, a warning is supplied:

```
exc-destr.cpp:6:13: warning: '~A' has a non-throwing exception
specification but can still throw [-Wexceptions]
            throw "Thrown by Destructor";
            ^
exc-destr.cpp:5:9: note: destructor has a implicit non-throwing
exception specification
        ~A() {
        ^
1 warning generated.
```

Exception in Destructor 3/3

Using `noexcept` spec we get predictable behavior:

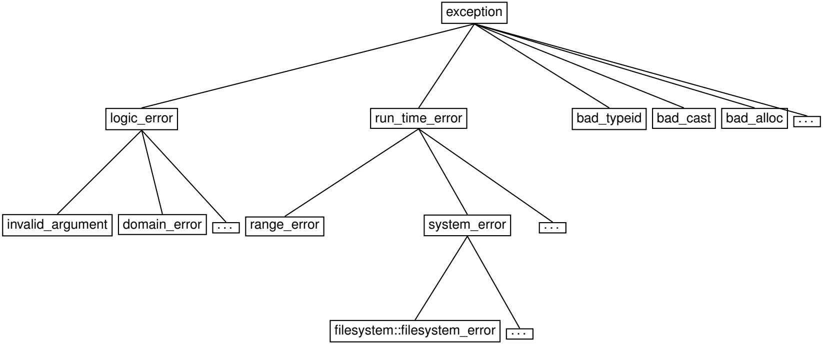
```
class A {
    public:
        ~A() noexcept(false) {
            throw "Thrown by Destructor";
        }
};
```

Output:

```
Print Thrown by Destructor
```

Plan

Standard Exceptions Hirerarchy (partial)



Example: regex-error (<regex>)

Source code:

```
try {
  std::regex re("[a-z]*[a");
}
catch (const std::regex_error& e) {
  std::cout << "regex_error caught: " << e.what() << '\n';
  if (e.code() == std::regex_constants::error_brack) {
    std::cout << "There is an error_brack.\n";
  }
}
```

Testing:

```
$ g++ regex-error.cpp -std=c++11
$ ./a.out
regex_error caught:
  The expression contained mismatched [ and ].
  There is an error_brack.
```

Example: bad-function-call (<functional>)

Source code:

```
std::function<int()> f = nullptr;
try {
  f();
} catch(const std::bad_function_call& e) {
  std::cout << "bad_function_call caught: " << e.what() << '
}
return 0;
```

Testing:

Mac OS

```
$ g++ bad-function-call.cpp -std=c++11
$ ./a.out
bad_function_call caught: std::exception
```

Windows:

```
> cl bad-function-call.cpp /std:c++14
> bad-function-call.exe
bad_function_call caught: bad function call
```

Example: bad-alloc (<new>) 1/3

Source code:

```
int negative = -1;
int small = 1;
int large = INT_MAX;
int opt = -1;
while (opt != 0) {
  try {
    printMenu();
    cin >> opt;
    switch (opt) {
    case 1: new int[negative]; break;
    case 2: new int[small]{1,2,3};  break;
    case 3: new int[large][1000000]; break;
    default: opt = 0;
    }
  }catch(const std::bad_array_new_length &e) {
    cout << "bad_array_new_length caught: " << e.what() << '\n';
  } catch(const std::bad_alloc &e) {
    cout << "bad_alloc caught: " << e.what() << '\n';
  } catch(...) {
    cout << "unknown exceptions caught.";
  }
}
```

Example: bad-alloc (<new>) 2/3

Testing Mac OS:

```
1. Negative.
2. Small.
3. Large.
Option: 1
bad_alloc caught: std::bad_alloc

1. Negative.
2. Small.
3. Large.
Option: 2
bad_alloc caught: std::bad_alloc

1. Negative.
2. Small.
3. Large.
Option: 3
a.out(21194,0x11e7b15c0) malloc: can't allocate region
*** mach_vm_map(size=8589934588002304) failed (error code=3)
a.out(21194,0x11e7b15c0) malloc: *** set a breakpoint in malloc_error_
bad_alloc caught: std::bad_alloc

1. Negative.
2. Small.
3. Large.
Option:
```

Example: bad-alloc (`<new>`) 3/3

Testing Windows:

```
1. Negative.
2. Small.
3. Large.
Option: 1
bad_array_new_length caught: bad array new length

1. Negative.
2. Small.
3. Large.
Option: 2

1. Negative.
2. Small.
3. Large.
Option: 3
bad_array_new_length caught: bad array new length
```

exception Class

```
namespace std {
  class exception {
  public:
    exception() noexcept;
    exception(const exception&) noexcept;
    exception& operator=(const exception&) noexcept
    virtual ~exception();
    virtual const char* what() const noexcept;
  };
}
```

Deriving from `exception`

```
class MyException : public exception {
 public:
   const char * what () const noexcept
   {
     return "Division by zero. ";
   }
};

int safeDiv(int divident, int divisor)
{
  if (divisor == 0)
    throw MyException();
  return divident / divisor;
}
```

Testing

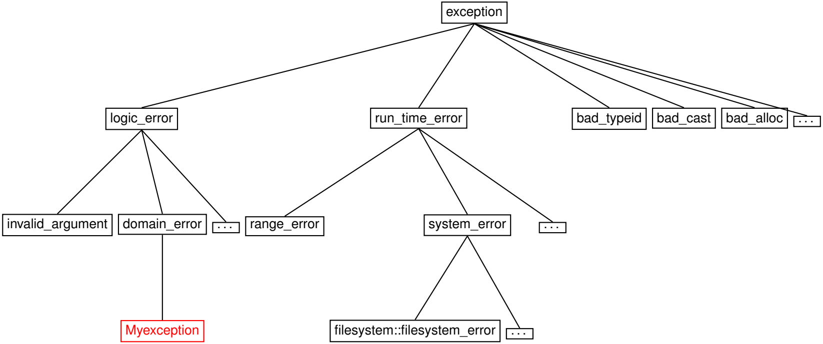
Source:

```
try {
    safeDiv(3, 0);
}  catch(MyException& exc) {
    std::cout << "MyException caught" << std::endl;
    std::cout << exc.what() << std::endl;
} catch(std::exception& e) {
    //Other errors
}
```

Output:

```
$ g++ demo.cpp
$ ./a.out
MyException caught
Division by zero.
```

Deriving from Standard Exceptions



Plan

`noexcept()` Operator

- tests whether or not an expression throws an exception
- returns `false` if there is any subexpression that can throw exceptions, i.e., if there is any expression that is not specified with `noexcep(true)` or `throw()`
- returns true if all subexpressions are specified with noexcept (true) or throw ()
- very useful when moving objects (e.g., `reserve()` method)

`noexcept()`: Example

```
class A {
public:
  A() noexcept(false) {
    throw 2;
  }
};

class B {
public:
  B() noexcept(true) {  }
};
```

noexcept () :: Testing 1/3

Source:

```
template <typename T>
void f() {
  if (noexcept(T()))
    std::cout << "NO Exception in Constructor" << std::e
  else
    std::cout << "Exception in Constructor" << std::endl
}

int main() {
  f<A>();
  f<B>();
}
```

Output:

```
$ g++ noexcept-operator.cpp -std=c++11
$ ./a.out
Exception in Constructor
NO Exception in Constructor
```

noexcept() :: Testing 2/3

Testing in a similar way the destructor does not work:

```
class C {
public:
  C() noexcept(true) {}
  ~C() noexcept(false) {}
};

template <typename T>
void g() {
  if (noexcept(~T()))
    std::cout << "NO Exception in Destructor\n";
  else
    std::cout << "Possible Exception in Destructor\n";
}
int main() {
  g<C>();
}
```

Output:

```
$ g++ -std=c++17 noexcept-declval.cpp
noexcept-declval.cpp:13:16: error: invalid argument type 'C' to
  if (noexcept(~T()))
               ^~~~
```

noexcept() :: Testing 3/3

Use `std::declval<T>()`:

```
class C {
public:
  C() noexcept(true) {}
  ~C() noexcept(false) {}
};

template <typename T>
void g() {
  if (noexcept(std::declval<T>().~T()))
    std::cout << "NO Exception in Destructor\n";
  else
    std::cout << "Possible Exception in Destructor\n";
}

int main() {
  g<C>();
}
```

Output:

```
$ g++ -std=c++17 noexcept-declval.cpp
$ ./a.out
Possible Exception in Destructor
```

noexcept can lie 1/2

```
class One :  public exception { };
class Two :  public exception { };
void g() {
  throw "Surprise.";
}
void fct(int x) noexcept {
  switch (x){
  case 1: std::cout << "ONE" << std::endl; return;
  case 2: std::cout << "TWO" << std::endl; return;
  }
  g();
}
```

`noexcept` can lie 2/2

Testing source:

```
int main(void) {
  if (noexcept(fct(3)))
    std::cout << "NO possible exception" << std::endl;
  else
    std::cout << "Possible exception" << std::endl;
  fct(3);
  return 0;
}
```

Output:

```
$ ./t.exe
NO possible exception
libc++abi: terminating due to uncaught exception of type char
zsh: abort      ./t.exe
```

C++2017: `noexcept` is a Part of Function Type

Source:

```
void g() noexcept {}
int main()
{
    auto f = g;
    std::cout << std::boolalpha  \
              << noexcept(f()) << std::endl;
}
```

Output:

```
$ g++ -std=c++17 exc-part-of-type.cpp
$ ./a.out
true
```

Recommendations

- always specify exceptions
- always start with standard exceptions
- declare the exceptions of a class within it
- uses exception hierarchies
- captures by references
- throw exceptions in constructors
- attention to memory release for partially created objects
- be careful how you test if an object has been created OK
- do not cause exceptions in destructors unless it is a must and then do it very carefully (preferably in C ++ 2017 or after)

Recommendations

Some concerns to care of when throw exceptions in constructors/destructors:

- if a constructor throws an exception when the stack is in an unstable state, then program execution ends
- it is almost impossible to design predictable and accurate containers in the presence of exceptions in destructors
- certain pieces of C ++ code may have undefined behavior when destructors throw exceptions
- what happens to the object whose "destruction" failed (because the destructor method threw an exception)?