Programare orientata- obiect (POO) utilizand C++

Exceptii

Dorel Lucanu

Erori

- “ … 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.”
  - Bjarne Stroustroup (parintele C++ului)

- Aceasta rata poate fi imbunatatita prin insusirea de tehnici de programare corecte

Trei cerinte esentiale pentru un program

- sa produca iesirile dorite (conform specificatiilor) pentru intrarile legale (corectitudine)
- sa dea mesaje de eroare rezonabile pentru intrarile nelegale
- sa permita terminarea in cazul gasirii unei erori

Tipuri de erori

- erori de compilare
  - depistate de compilator
  - in general usor de fixat
- erori de editare a legaturilor
  - functii/metode neimplementate
  - lispsa de biblioteci
- erori in timpul executiei
  - sursa poate fi calculatorul, o componenta dintr-o biblioteca, sau programul insusi
- erori logice
  - programul nu are comportarea dorita
  - pot fi detectate de programator (prin testare)
  - … sau de utilizator (client)

Erorile trebuie raportate

- posibila aparitie a unei erori trebuie detectata si raportata

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

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

Cum se raporteaza o eroare?

- raportarea de erori trebuie sa fie uniforma: acelasi mesaj pentru acelasi tip de eroare (pot diferi informatiile de localizare)
- de aceea trebuie gestionat un “indicator de eroare”
- asocierea de numere pentru erori, este o solutie dar poate fi problematica
- POO are mijloacele necesare pentru a organiza inteligent depistarea si raportarea de erori

Eroare sau exceptie?

- de multe ori cei doi termeni sunt confundati
- exista totusi o deosebire (subtila) intre intelesurile celor doua notiuni
- eroare = greseala depistata in timpul functionarii ce trebuie eliminata prin repararea programului
- exceptie = comportare neprevazuta care poate aparea in situatii rare sau foarte rare
- exceptiile trebuie tratate in cazul programelor

- o exceptie netratata constituie o eroare

- in acest curs discutam mai mult despre exceptii

Exceptii in POO: scop

- Exceptiile in POO au fost proiectate cu intentia de a separa logica business de mecanismul de transmitere a erorilor
- Scopul este de a permite manipularea erorilor, care apar ca exceptii, la un nivel potrivite ce nu intereferează cu logica business

Un prim studiu de caz simplu

- Implementarea unui tip de data abstract

- Reamintire:
  - **tip de data abstract** = o descriere a unui tip de data independent de reprezentarea datelor si implementarea operatiilor

  - O **clasa poate fi utilizata pentru a implementa un tip de date abstract**. Ea defineste atribute si metode care implementeaza structura de date respectiv operatiile tipului de date abstract.

Stiva

- tipul de data abstract Stiva
  - entitati de tip data: liste LIFO
  - operatii
    - ⇨ empty()
    - ⇨ push()
    - ⇨ pop()
    - ⇨ top()
    - ⇨ isEmpty()

Stiva.h

```
template <class T>
class Stack
{
public:
  Stack();     // implementeaza empty()
  ~Stack();
  void push(T);
  void pop();
  T top();
  bool isEmpty();
private:
  T elt[MAX_STACK];
  int topIndex;
};
// …
```

O prima implementare pentru push, top

```
template <class T>
void Stack<T>::push(T x)
{
  elt[++topIndex] = x;
}

template <class T>
T Stack<T>::top()
{
  return elt[topIndex];
}
```

Testare

```
#define MAX_STACK 5
```

```
int i; char c = 'a';
Stack<char> st;
for (i = 0; i <= 9; ++i) {
  st.push(c++);
  cout << st.top() << endl;
}
```

Erorile nu-s usor de gasit intotdeauna

```
$ g++ test-stack.cpp -o test-stack.exe
$ ./test-stack.exe
a
b
c
d
e
f
g
h
?
j
```

Erorile nu-s usor de gasit intotdeauna

```
$ g++ test-stack.cpp -o test-stack.exe
$ ./test-stack.exe
a
b
c
d
e
f
g
h
?
j
```

Testare

```
#define MAX_STACK 5
```

```
for (i = 0; i <= 9; ++i) {
    st.push(c++);
    //      cout << st.top() << endl;
  }
  Stack<char> st2 = st;
  cout << st2.top() << endl;
```

Erorile nu-s usor de gasit intotdeauna

```
$ g++ test-stack.cpp -o test-stack.exe
$ ./test-stack.exe

$
```

Care e sursa erorii?

Sursa erorii

- stiva are o capacitate marginita de MAX_STACK
- de aceea oparatia push() este partial definita; nu se poate introduce un element daca stiva este plina
- operatiile top() si pop() sunt si ele operatii partiale: nu se poate citi/elimina un element din stiva vida
- sursa eroriii poate fi in proiectarea metodelor sau in utilizarea lor
- utilizatorul (clientul) nu cunoaste in general capacitatea stivei

Exigent sau tolerant?

- cui revine responsabilitatea de a verifica daca sunt satisfacute conditiile de apelare corecta ale operatiilor?
- proiectarea unei clase/metode e de natura “contractuala” intre clasa/metoda (furnizorul de servicii) si utilizatorul (clientul)
- design by contract – concept introdus de Bertrand Meyer
  - “if you give me a state satisfying the precondition, I give you a state satisfying the postcondition”
  - preconditia = proprietatile ce trebuie sa le satisfaca datele de intrare
  - postconditia = proprietatile ce trebuie sa le satisfaca datele de iesire
- exigent – verificarea preconditiei se face de catre client
- tolerant – verificarea preconditiei se face de catre metoda

Versiunea “tolerant”

- e discutabil care varianta e mai buna – exigent sau tolerant (depinde pe cine intrebi: clientul sau furnizorul …)
- daca optam pe varianta “tolerant”, atunci aparitia exceptiilor trebuie raportata
- se poate face aceasta raportare in mod sistematic?
- in POO da, utilizand mecanismul de management al exceptiilor

Metodele push() si top() revizuite

```
template <class T>
void Stack<T>::push(T x)  {
  if (topIndex == MAX_STACK)
    throw  "Class Stack overflow.";
  elt[topIndex++] = x;
}

template <class T>
T Stack<T>::top() {
  if (topIndex < 0)
    throw "Try reading from an empty stack.";
  return elt[topIndex];
}
```

“Aruncarea” de exceptii nu e suficienta ...

```
$ g++ test-stack.cpp -o test-stack.exe
$ ./test-stack.exe
a
b
c
d
e
libc++abi.dylib: terminating with uncaught exception of type char const*
Abort trap: 6
```

- exceptiile “aruncate” trebuie sa fie si “prinse”

Structurile try - catch

- codul susceptibil de a arunca exceptii se include intr-un bloc “try”

```
try {
    for (i = 0; i <= 9; ++i) {
      st.push(c++);
      cout << st.top() << endl;
    }
  }
```

nivelul cu logica business

- codul care trateaza un anumit tip de exceptii se include intr-un bloc “catch”, aflat dupa blocul “try”

```
catch(char const* msg) {
    cout << msg << endl;
  }
```

nivelul cu manipularea erorilor/exceptiilor

Testare

```
$ g++ test-stack.cpp -o test-stack.exe
$ ./test-stack.exe
a
b
c
d
e
Class Stack overflow.
```

Clauze catch() multiple

- exceptiile pot fi clasificate in mai multe tipuri
  - exceptie stiva plina

```
template <class T>
class StackOverflowException {
 public:
  StackOverflowException(Stack<T>);
  void debugPrint();
 private:
  Stack<T> stackErr;
};
```

Clauze catch() multiple

- exceptie stiva vida

```
template <class T>
class StackEmptyException {
 public:
  void debugPrint();
};
```

- exceptie impartire prin zero

```
class DivByZeroException {
 public:
  DivByZeroException(int);
  void debugPrint();
 private:
  int divident;
};
```

Clauze catch() multiple

- definitia metodei debugPrint() (doar un caz, celelalte sunt similare)

```
template <class T>
void StackOverflowException<T>::debugPrint()
{
  Stack<T> stCopy = stackErr;
  std::cout << "Stack Overflow exception." << std::endl;
  while ( ! stCopy.isEmpty() ) {
    std::cout << stCopy.top() << std::endl;
    stCopy.pop();
  }
}
```

Clauze catch() multiple

- alte doua functii care arunca exceptii

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

void unknown()
{
  throw rand();
}
```

Clauze catch() multiple

- blocul try …

```
try {
   switch (option) {
     case 1:
       for (i = 0; i <= 9; ++i) st.push(c++);
     case 2:
       for (i = 0; i <= 9; ++i) st.pop();
     case 3:
       safeDiv(3, 0);
     case 4:
       unknown();
     default:
    cout << "Execution without exceptions." << endl;
   }
}
```

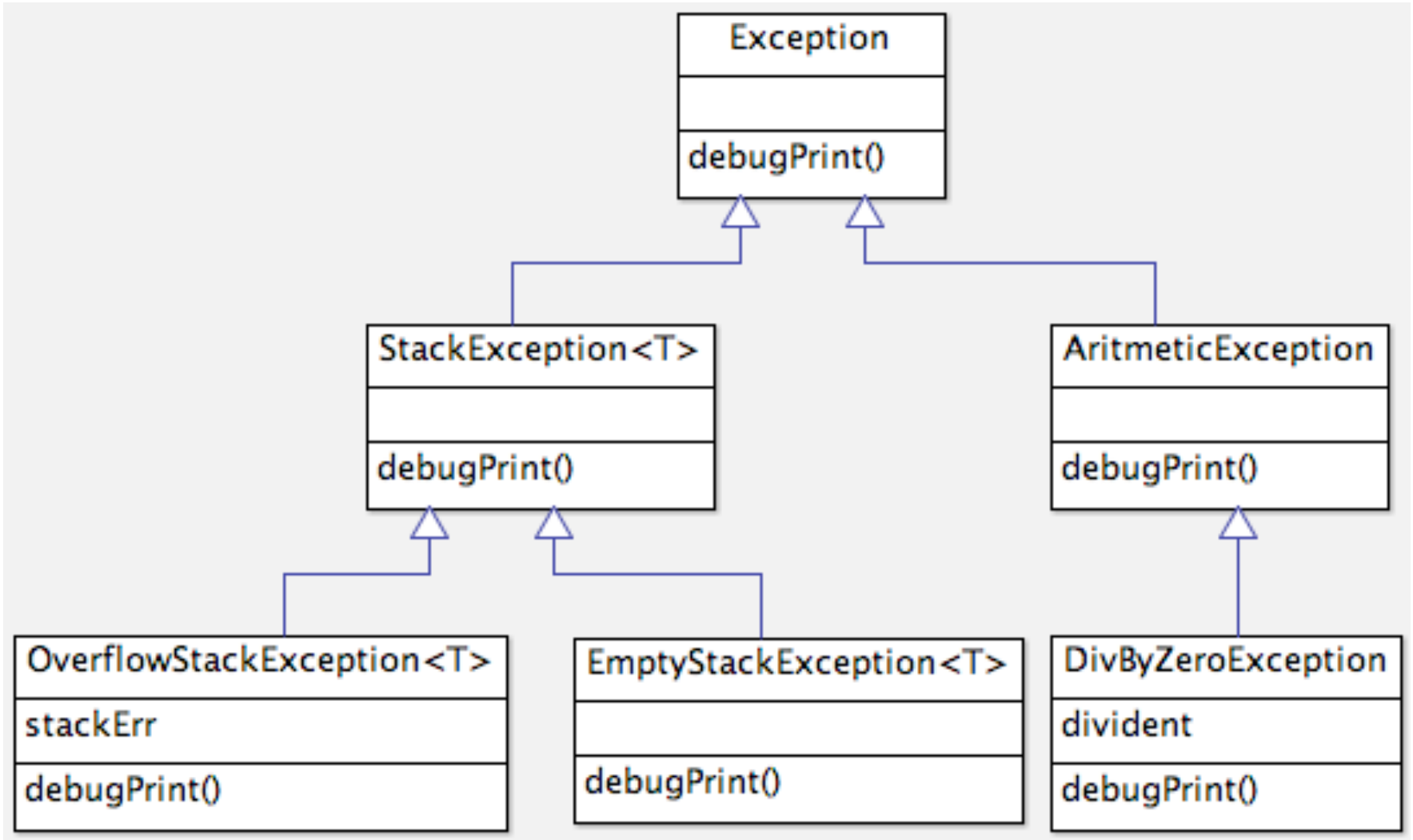
Clauze catch() multiple

- … urmat de clauzele catch()

```
catch(StackOverflowException<char> excpt) {
  excpt.debugPrint();
}
catch(StackEmptyException<char> excpt) {
  excpt.debugPrint();
}
catch(DivByZeroException excpt) {
  excpt.debugPrint();
}
catch(...) {
  cout << "Unknown exception." << endl;
}
```

Demo

Ierarhii de exceptii



Ierarhii de exceptii

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

template <class T>
class StackException : public Exception {
 public:
  virtual void debugPrint() {
    this->Exception::debugPrint();
    std::cout << "Stack:";
  }
};
```

Ierarhii de exceptii

```
template <class T>
class StackOverflowException : public StackException<T>  {
 public:
  StackOverflowException(Stack<T>);
  virtual void debugPrint();
 private:
  Stack<T> stackErr;
};

template <class T>
class StackEmptyException : public StackException<T> {
 public:
  virtual void debugPrint();
};
```

Ierarhii de exceptii

```
class ArithmeticException : public Exception {
 public:
  void debugPrint() {
    this->Exception::debugPrint();
    std::cout << "Arithmetic:";
  }
};

class DivByZeroException : public ArithmeticException {
 public:
  DivByZeroException(int);
  void debugPrint();
 private:
  int divident;
};
```

Ierarhii de exceptii

```
template <class T>
void StackOverflowException<T>::debugPrint()
{
  this->StackException<T>::debugPrint();
  std::cout << "Overflow." << std::endl;
  Stack<T> stCopy = stackErr;
  while ( ! stCopy.isEmpty() ) {
    std::cout << stCopy.top() << std::endl;
    stCopy.pop();
  }
}

// celelalte sunt definite similar
```

Totusi comportarea nu cea chiar dorita

```
try {
   switch (option) {
     // the same
   }
 }
 catch(Exception excpt) {
   excpt.debugPrint();
 }

catch(...) {
   cout << "Unknown exception." << endl;
 }
```

Totusi comportarea nu cea chiar dorita

- $ g++ demo.cpp
- $ ./a.out
- Option: 1
- Exception: $ ./a.out
- Option: 2
- Exception: $ ./a.out
- Option: 3
- Exception: $

- aceasta pentru ca parametrul lui catch() este transmis prin copiere si de aceea se pierde comportarea polimorfica

Comportarea dorita e obtinuta cu referinte

```
try {
   switch (option) {
     // the same
   }
 }
catch(Exception& excpt) {
   excpt.debugPrint();
 }

catch(...) {
   cout << "Unknown exception." << endl;
 }
```

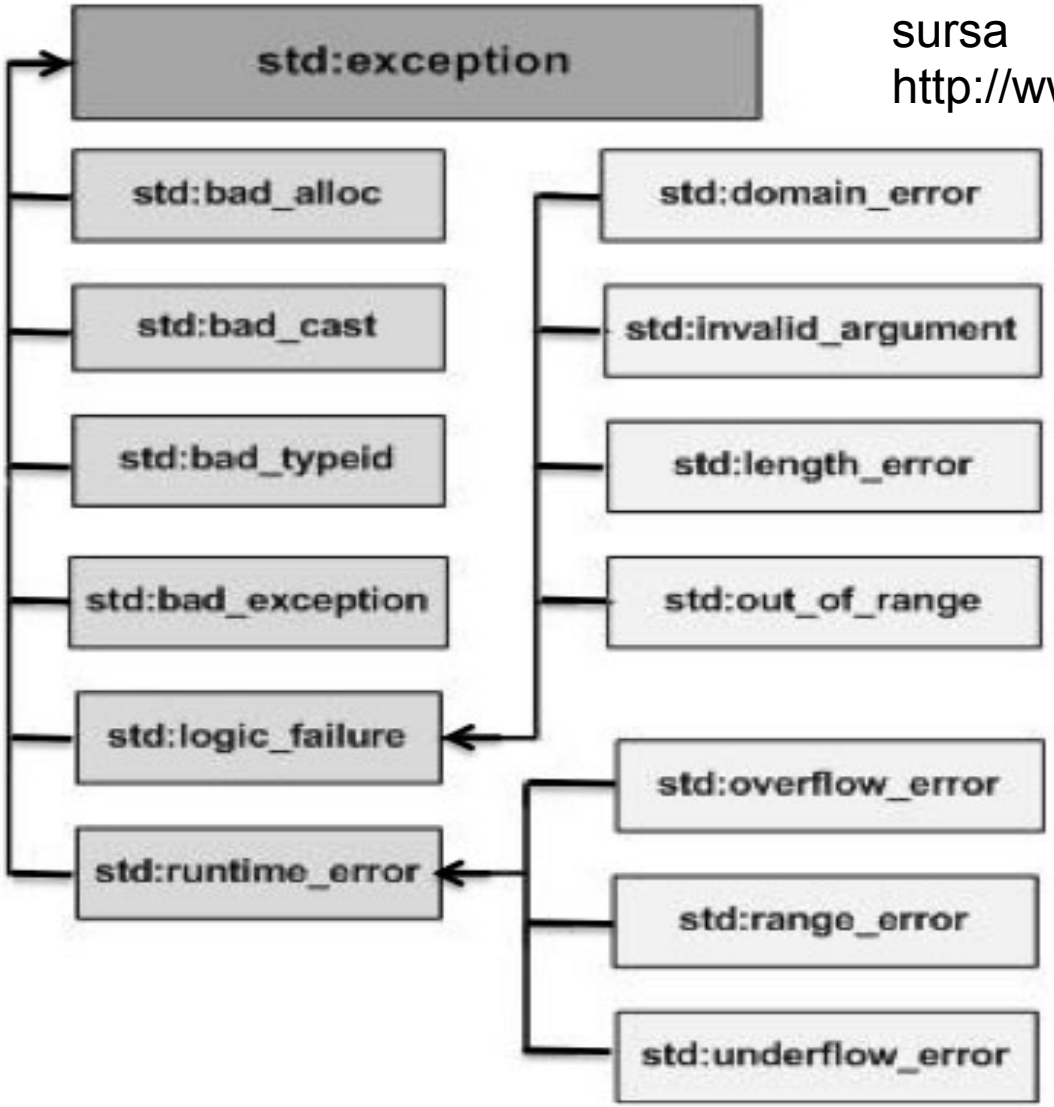
Totusi comportarea nu cea chiar dorita

```
$ g++ demo.cpp
$ ./a.out
Option: 1
Exception: Stack:Overflow.
e
d
c
b
a
$ ./a.out
Option: 2
Exception: Stack:Empty.
$
```

Exceptiile pot face parte din specificatie

```
template <class T>
class Stack
{
public:
  Stack();
  ~Stack();
  void push(T) throw(StackException<T>);
 //…
}
template <class T>
void Stack<T>::push(T x) throw(StackException<T>)
{
  // the same
}
```

Exceptiile standard C++ (<exceptions>)



sursa
http://www.tutorialspoint.com/cplusplus

Clasa “exception”

```
class exception {
public:
  exception () throw();
  exception (const exception&) throw();
  exception& operator= (const exception&) throw();
  virtual ~exception() throw();
  virtual const char* what() const throw();
  // …
}
```

Derivare din “exception”

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

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

Testare

```
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
}
```

Testare

```
$ g++ demo.cpp
192-168-0-102:inherit-from-exception dlucanu$ ./a.out
MyException caught
Division by zero.
```

Testare

```
try
 {
   throw(1);
 }
 catch(MyException& exc)
 {
   std::cout << "MyException caught" << std::endl;
   std::cout << exc.what() << std::endl;
 }
 catch(std::exception& e)
 {
   //Other errors
 }
```

Testare

```
$ ./a.out
libc++abi.dylib: terminating with uncaught exception of type int
Abort trap: 6
```

Functia `unexpected()`

- Din manual:

“Function handling unexpected exceptions

Calls the current unexpected handler.

By default, the unexpected handler calls terminate. But this behavior can be redefined by calling set_unexpected.

This function is automatically called when a function throws an exception that is not listed in its dynamic-exception-specifier (i.e., in its throw specifier).

This function is provided so that the unexpected handler can be explicitly called by a program, and works even if set_unexpected has not been used to set a custom unexpected handler (calling terminate in this case).”

Functia unexpected()

```
class One : public exception { };
class Two : public exception { };
void g()
{
  throw "Surprise.";
}
void fct(int x) throw (One, Two)
{
  switch (x){
      case 1: throw One();
      case 2: throw Two();
  }
  g();
}
```

Functia unexpected()

```
void my_unexpected() {
   cout << "My unexpected exception.";
   exit(1);
}
int main(){
   set_unexpected(my_unexpected);
   int option;
   cout << "Option (1-3): "; cin >> option;
   try {
      fct(option);
   } catch (One) {
      cout << "Exception one" << endl;
   } catch (Two)  {
      cout << "Exception two" << endl;
   }
   return 0;
}
```

Testare

```
$ g++ unexpected.cpp
$ ./a.out
Option (1-3): 1
Exception one
$ ./a.out
Option (1-3): 2
Exception two
$ ./a.out
Option (1-3): 3
My unexpected exception.
```

Functia `terminate()`

- Din manual:

“Function handling termination on exception

Calls the current terminate handler.

By default, the terminate handler calls abort. But this behavior can be redefined by calling set_terminate.

This function is automatically called when no catch handler can be found for a thrown exception, or for some other exceptional circumstance that makes impossible to continue the exception handling process.

This function is provided so that the terminate handler can be explicitly called by a program that needs to abnormally terminate, and works even if set_terminate has not been used to set a custom terminate handler (calling abort in this case).”

Functia `terminate()`

- exemplu: exceptions_nested.cpp

```
class Unu   { };
class Doi   { };
class Trei  { };
class Patru { };
```

```
void unu() {
        throw Unu();
}
void doi() {
        throw Doi();
}

void trei() {
        throw Trei();
}

void patru() {
        throw Patru();
}
```

Functia `terminate()`

- exemplu: exceptions_nested.cpp (continuare)

```
void my_terminate()
{
  cout << "Revin in 5 min!" << endl;
  abort();
}

void (*old_terminate)()
  = set_terminate(my_terminate);
```

Functia `terminate()`

```
try {
    try   {
          try   {
                // unu();
                // doi();
                // trei();
                patru();
          } catch (Trei) {
                cout << "Exceptie Trei.";
          }
    } catch (Doi) {
          cout << "Exceptie Doi.";
    }
} catch(Unu) {
    cout << "Exceptie Unu.";
}
```

Testare functia `terminate()`

```
$ g++ exceptions_nested.cpp
$ ./a.out
Revin in 5 min!
Abort trap: 6
```

Exceptii::constructori

- exceptiile pot aparea si in constructori
- trebuie avut grija de obiectele create partial

```
class AA
{
public:
   AA(const char* s="\0") throw(int);
   ~AA();
private:
   static int nrOb;
   int id;
   char* nume;
};
```

Exceptii::constructori

```
AA::AA(const char* s) throw(int) {
   int n = strlen(s);
   nume = new char[n+1];
   strcpy(nume, s);
   nrOb++;
   id = nrOb;
   cout << id << " AA() "<< s << endl;;
   if (nrOb == 3) throw int(3);
   if (isdigit(s[0])) throw char(*s);
}

AA::~AA() {
   cout << id << " ~AA()\n";
   delete [] nume;
   nrOb--;
}
```

Exceptii::constructori

```
void my_unexpected()
{
  cout << "Exceptie neprevazuta.\n";
  throw;
}
void my_terminate()
{
  cout << "Revin in 5 min.\n";
  exit(1);
}

int AA::nrOb = 0;
```

Exceptii::constructori

```
int main() {
   set_unexpected(my_unexpected);
   set_terminate(my_terminate);
   try {
       AA a("start");
       AA* b = new AA[5];
       AA c("stop");
   } catch (int i) {
       cout << "Exceptie: " << i << endl;
   }
   try {
       AA d("1234");
   } catch (char c) {
       cout << "A fost aruncat " << c << endl;
   }
  return 0;
}
```

Exceptii::constructori

- testare

```
1 AA() start
2 AA()
3 AA()
2 ~AA()
1 ~AA()
Exceptie: 3
2 AA() 1234
Exceptie neprevazuta.
Revin in 5 min.
```

Exceptii in destructori?

- Cateva motive pentru care nu se recomanda aruncarea de exceptii de catre constructori:
  - daca un constructor arunca o exceptie cand stiva este intr-o stare instabila, atunci executia programului se termina
  - este aproape imposibil sa proiectezi containere predictibile si corecte in prezenta exceptiilor in destructori
  - anumite piese de cod C++ pot avea un comportament nedefinit cand destructorii arunca exceptii
  - ce se intampla cu obiectul a carui “distrugere” a esuat (din cauza ca metoda destructor a aruncat o exceptie)?

Exceptii in destructori?

- Stroustrup: "the vector destructor explicitly invokes the destructor for every element. This implies that if an element destructor throws, the vector destruction fails... There is really no good way to protect against exceptions thrown from destructors, so the library makes no guarantees if an element destructor throws" (from Appendix E3.2)

Exceptii in destructori: predictibil

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

Exceptii in destructori: predictibil

```
$ g++ destr.cpp
$ ./a.out
Print Thrown by Destructor
$
```

Exceptii in destructori: impredictibil

```
class A {
    public:
        ~A() {
            throw "Thrown by Destructor";
        }
};
int main() {
    try {
        A   a; throw 2;
    }
    catch(...){
        std::cout << "Never print this “;
    }
}
```

Exceptii in destructori: predictibil

```
$ g++ destr.cpp
$ ./a.out
libc++abi.dylib: terminating with uncaught exception of
type char const*
Abort trap: 6
```

C++2011

- Specificatie “throw” vida
  - C++ 2003

  void f() throw();
  - C++ 2011

  void f() noexcept(true);

C++2011

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

Exceptii in destructori: predictibil

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

Declaratia destructorului este echivalenta cu

```
~A() noexcept(true) {
  throw "Thrown by Destructor";
}
```

C++2011

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

Exceptii in destructori: predictibil

```
$ g++ destr.cpp -std=c++11
$ ./a.out
Print This Thrown by Destructor
$
```

Operatorul noexcept()

- testeaza daca o expresie arunca sau nu o exceptie
- intoarce *false* daca exista vreo subexpresie poate arunca exceptii, adica daca exista vreo expresie care nu este specificata cu noexcept(true) ori throw()
- intoarce *true* daca toate subexpresiile sunt specificate cu cu noexcept(true) ori throw()
-
- foarte util la mutarea de obiecte (e.g., metoda reserve())

Operatorul noexcept()

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

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

Operatorul noexcept()

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

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

Operatorul noexcept()

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

Operatorul noexcept()

- Testarea daca la constructia sau distrugerea unui obiect de tip e posibil sa se arunce exceptii

noexcept(T(std::declval<T>()))

- functia declval() converteste un tip T la un tip referinta (rvalue)

```
template <typename T>
  typename std::add_rvalue_reference<T>::
     type declval() noexcept;
```

Operatorul noexcept()

```
class C {
public:
  C() noexcept(true) {}
  ~C() {}
}
template <typename T>
void g() {
  if (noexcept(T(std::declval<T>())))
    std::cout << "NO Exception in Any
Constructor or Destructor”;
  else
    std::cout << "Possible Exception in a
Constructor  or Destructor”;
}
```

Operatorul noexcept()

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

```
$ g++ noexceptop.cpp -std=c++11
$ ./a.out
NO Exception in Any Constructor or Destructor
$
```

Operatorul noexcept()

```
class C {
public:
  C() noexcept(true) {}
  C(const C&) {}
  ~C() {}
}
template <typename T>
void g() {
  if (noexcept(T(std::declval<T>())))
    std::cout << "NO Exception in Any
Constructor or Destructor”;
  else
    std::cout << "Possible Exception in a
Constructor  or Destructor”;
}
```

Operatorul noexcept()

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

```
$ g++ noexceptop.cpp -std=c++11
$ ./a.out
Possible Exception in a Constructor or Destructor
$
```

Operatorul noexcept()

```
class C {
public:
  C() noexcept(true) {}
  ~C() noexcept(false) {}
}
template <typename T>
void g() {
  if (noexcept(T(std::declval<T>())))
    std::cout << "NO Exception in Any
Constructor or Destructor”;
  else
    std::cout << "Possible Exception in a
Constructor  or Destructor”;
}
```

Operatorul noexcept()

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

```
$ g++ noexceptop.cpp -std=c++11
$ ./a.out
Possible Exception in a Constructor or Destructor
$
```

Exceptii::recomandari

- specifica totdeauna exceptiile
- porneste totdeauna cu exceptiile standard
- exceptiile unei clase declara-le in interiorul ei
- utilizeaza ierarhii de exceptii
- captureaza prin referinte
- arunca exceptii in constructori
  - atentie la eliberarea memoriei pentru obiectele create partial
  - atentie cum testezi daca un obiect a fost creat OK
- nu cauza exceptii in destructori decat daca e musai si atunci cu mare atentie (de preferat in C++ 2011 sau dupa)