Programmazione Avanzata per il Calcolo Scientifico
Advanced Programming for Scientific Computing
Lecture title: Exception Handling

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

Preconditions, postconditions and invariants

A class or a function may be thought as a map from data given in input to data provided as output. A specification which characterizes a valid input data is called precondition, and it is decided by the developer of the class/function who also should guarantee that the expected output (postcondition) is provided whenever the input respect the precondition. Failure to do so is a fault (bug) in the code.

An invariant of the class is a condition that should be satisfied by the state of an object at any time (excluding transient situations like the construction process). A code is in an inconsistent state if invariants are not satisfied.

Testing pre and post conditions, as well as invariants, is part of code verification during the development phase.

An example

```
Matrix Chowlesky(Matrix const & m);
```

This function has a precondition that matrix m is symmetric positive definite. The postcondition is that the matrix in output is a lower triangular matrix representing the Chowlesky factorization of m.
A symmetric matrix m has as invariant that m(i,j)=m(j,i) for all matrix elements.

What is an exception

An exception is an anomalous conditions that would change the normal flow of program execution if not handled. It is not a bug: the special condition is not related to incorrect coding but to situations difficult or impossible to foresee.

Examples: no memory available after a `new` call. Failure to open a file due to insufficient privileges. An invalid floating point operation (floating point exception (FPE)) that cannot be easily detected “a-priori”.

Note: an incorrect behaviour (like failure to satisfy a postcondition for a correct input data) that is related to incorrect coding is not an exception. These conditions (called bugs) should be found (possibly!) during the development phase, maybe with the help of `assert` or more sophisticated debugging techniques.

Why to handle exceptions?

In the old days, particularly in the scientific computing context, exceptions where not handled at all or, at most, they would abort the program with an error message.

With the advent of graphical interface, the handling of exceptions has become more important since one does not want that the failure of an algorithm causes the abort of the program. Besides, there are other context where handling of the exceptions is necessary also in scientific programming.

Exception handling in C++

C++ introduces an effective mechanism to handle exceptions (further bettered in C++11). The basic mechanism is rather simple:

- If an exception occurs the program uses the command **throw** to "throw" an "exception": an object decided by the programmer that contains information about the exception;
- **throw** causes the calling function and all the function in the call stack to return unless the exception is "caught" by a **try**{} **catch**() block.

example

```
class Error{...};// a user defined class
void f() {
...
if (exception-condition) {
throw Error(); // throw an object of class Error
}
...}
...
int main()
{
try {...
f();
...}
catch (const Error& e) { do something}
```

throw, try and catch

Command **throw** ̀is followed by an object (that may be built on the spot) and can by basically anything (a string, an int, a object of user defined type). The object has the role of indicating that the exception has occurred and normally carries some useful information about the exception. The construct **try**{}**catch**{} is used to verify if and which exception has occurred. We may have

```
catch{Type}{...}
```

which captures exceptions of type Type, or

```
catch{const Type& e}{...}
```

which in addition allows to access the object that has been thrown, and

```
catch(...){}
```

which catches all exceptions that have been thrown (it is call the "catch all" clause). Polymorphism applies: **catch** (Base & x) captures all exception of type Base and derived from Base.

The **throw**() declaration

The declaration **throw**(Exc1, Exc2,...) at the end of a function definition was used to declare which exceptions were possibly thrown by the function in C++98. In C++11 it has been deprecated, so we omit to describe it further.
C++11 has instead introduced the operator noexcept that performs a compile-time check that returns true if an expression is declared to not throw any exceptions. Its use is not compulsory, but it may help the compiler to produce better code. More info in en.cppreference.com.

```
void fun1(double &, int) noexcept;// exception-free
class pippo{
 public:
 pippo();// May throw exceptions
 double f2(double) noexcept;// exception-free
};
```

Exception-safe and exception free

A piece of code is said to be exception-safe, if run-time failures within the code will not produce ill effects, such as memory leaks, garbled stored data, or invalid output. Exception-safe code must return the code to a *consistent state* after an exception has occurred (possibly the state before the exception).

A code is exception-free if it is guaranteed to satisfy its requirements with all possible input data. Es. all arithmetic operation on native (non-class) data are exception-free.

The design of a exception-free code requires to be able to *handle* the possible exception and take the correct action.

Standard exceptions

The Standard library provides some predefined exception classes, accessible with the header <exception>. They all derive form std::exception which is a struct that defines

```
virtual char const * what()const
```

a method that returns a message.
Many standard exceptions accepts a **char** * or a string (since c++11) containing the message in the constructor. Their usage is thus very simple

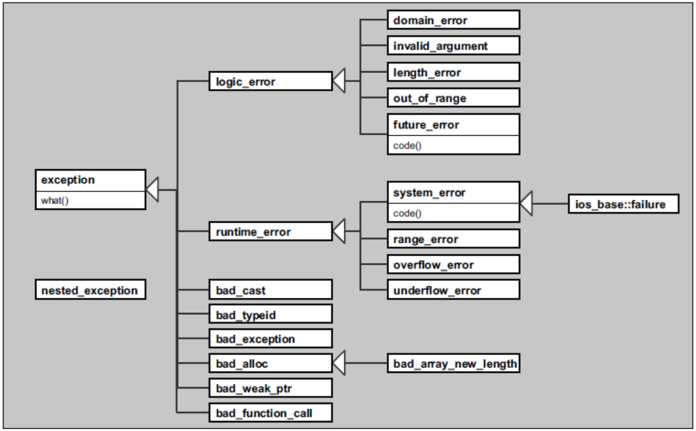
```
throw std::runtime_error(``A message'');
```

You may use the std exceptions directly or derive your class from them, for instance to redefine what():

```
 struct MyError: public std::exception
{ virtual char const * what()const
      {return "A stange error has occurred";}
};
 try {
 // some code here
}
 catch (int param) { cout << "int exception"; }
 catch (MyError & e) { cout << e.what(); }
 catch (...) { cout << "Some other exception"; }
```

Even if not compulsory, it is suggested to use the standard exceptions or to derive from a standard exception when creating a user defined one.

An overview of standard exceptions



Note: In C++11 some standard exceptions can also carry some user defined code. The handling of exceptions has been greatly enhanced in C++11 (exception pointers, nested exceptions...), it is beyond the scope of this lectures to give the details.

Some comments

Exception handling is now becoming an important issues in codes that must be integrated in a more general workflow or in a graphical interface. However, remember that the **try−catch** clause introduce inefficiencies: the code has to check if an exception has been thrown every time a function is called. High performance code usually makes little use of exception handling.

However, in many practical contests exception handling becomes compulsory. In that case you'd better use the noexept statement to indicate functions and methods that do not throw exceptions. This helps efficiency since the compiler will not check if an exception has been thrown when calling those functions.

An old style error control

If the failure of an algorithm is one of its possible outcome (for instance the failure of convergence of an iterative method), instead of throwing an exception one may return a status.

```
struct NewtStatus{
bool good;
int iterations;
double residual;
double lastvalue;}
```

```
NewtStatus Newton(double initial,const Fun & f);
```

However in this case it is better (simpler) if the error condition (good==**false**) is handled by the caller, not higher up in the calling stack.

Floating point exception

A particular attention should now be dedicated to possible exceptions arising from incorrect floating point operations. The issue here is that by default they do not throw exceptions. And this for obvious efficiency reason.

However, an invalid floating point operation can most of the times be detected (not everytime because it depends on the hardware!). But you must use the appropriate tools.

The IEEE standard establishes some error conditions (called floating point exceptions) (FPE) for floating point operation that should be detected by the processors, but by default they do not throw exceptions:

Invalid	Operations with mathematically invalid operands–for example, 0.0/0.0, sqrt(-1.0), and log(-37.8)
Division by zero	Divisor is zero and dividend is a finite nonzero number, es 9.9/0.0
Overflow	Operation produces a result that exceeds the range of the exponent
Underflow	Operation produces a result that is too small to be represented as a normal number
Inexact	Operation produces a result that cannot be represented with infinite precision–for example, 2.0 / 3.0

We will now see how to trap those exceptions.

Beware

Not all processors, in particular GPUs, are compliant with IEEE standard.

Also the activation of some aggressive optimization options, like `-ffast-math` can disable the possibility of trapping some FPE's. In some compilers the possibility of trapping FPEs is activated by a special compiler option (not in g++ nor clang).

Floating point environment

The floating-point environment (FENV) is a set of status flags and control modes supported by the implementation. Floating-point operations modify the floating-point status flags to indicate abnormal results or auxiliary information. The state of floating-point control modes affects the outcomes of some floating-point operations.
The C++11 standard establishes that FENV is only meaningful when **#pragma** STDC FENV_ACCESS is set to ON. Otherwise, the implementation is free to assume that floating-point control modes are always the default ones and that floating-point status flags are never tested or modified. In practice, few current compilers support the **#pragma** explicitly, but most compilers allow meaningful access to the floating-point environment anyway.

Handling FPEs

You need to include <cfenv> (<fenv.h> in C++98 where for the FPE support you had to use C). The following flags are defined (in fact there are more see here).

FE_INVALID	Invalid operations
FE_DIVBYZERO	Division by zero
FE_OVERFLOW	Overflow
FE_UNDERFLOW	Underflow
FE_INEXACT	Inexact operation
FE_ALLEXCEPT	All exceptions

These flags may be tested, or you may force the program to abort if they are raised (this later option is compiler dependent!).

Aborting on FPE

Using gnu or clang compilers on Unix systems to trap FPE at system level it is sufficient to call the function feenableexcept(FPEflags), where FPEFlags are the flags on which you want to activate the trap (you need <cfenv>).
Example:

```
feenableexcept(FE_INVALID|FE_DIVBYZERO|FE_OVERFLOW);
```

If a FPE of the type indicated by the flags happens the program emits a SIGFPE signal which is handled by the operative system and aborts the program.

Testing the Floating Point Environment

Maybe you do not want to abort the program, but only to test if a FPE has occurred. We have two main utilities to interact with the floating point environment.

```
int  fetestexcept(int FEFlag);
```

returns the bitwise OR of the status of the flag(s) FEFlag

```
void feclearexcept(FEFlags);
```

clears the FEFlags indicated.
Recall the when a flag is raised it remains so throughout the program unless you clear it.

An example of function that launches exceptions if an FPE occurs

```
#include <cfenv>
 void test_fpe_exception(){
using namespace std;
auto set_excepts =
       fetestexcept(FE_INVALID|FE_DIVBYZERO|FE_OVERFLOW);
  feclearexcept(FE_INVALID |FE_DIVBYZERO|FE_OVERFLOW);
  if (set_excepts & FE_INVALID) throw InvalidFPOperation();
  if (set_excepts & FE_OVERFLOW) throw FloatOverflow();
  if (set_excepts & FE_DIVBYZERO) throw ZeroDivision();
}
```

Here InvalidFPOperation etc are user defined exception whose complete definition is in FPExceptions/myexceptions.cpp.

Abort on FPE

Let assume that we want that our program aborts if a certain set of floating point exceptions occurs. But, since it causes a big overhead we want to de-activate it on "production codes". To do so we decide to activate the feature only if we compile the main program with the −DFPE_ABORT compiler option. A first possibility is to write in the main program

```
#ifdef FPE_ABORT
#include <cfenv>
#endif
 int main(){
#ifdef FPE_ABORT
 feenableexcept (FE_INVALID|FE_DIVBYZERO|FE_OVERFLOW);
 ...
#endif
 }
```

A more interesting implementation

The previous implementation is fine but it requires to write instructions in the main program. It would be nice to have the possibility that just by including a header file, let's call it trapfpe.hpp we have the activation of abort on FPE if the cpp macro FPE_ABORT is set.

This however raises a problem: functions con be normally executed only within the scope of the main() program. Then, how can we execute feenableexcept() from `trapfpe.hpp`, which will be included outside the scope of the main program?

We need to use a special compiler directive to tell the compiler:"this function must be loaded and executed before the main program starts". On gnu compilers and on clang++ the directive is __attribute__ ((constructor)).

A possible solution

In `trapfpe.hpp`

```
#ifdef FPE_ABORT
#include <cfenv>
struct FpeTrap{
  static void __attribute__ ((constructor))
  trapfpe (){
    /*! Enable some exceptions.*/
  std::feenableexcept (
             FE_INVALID|FE_DIVBYZERO|FE_OVERFLOW);}  };
 #endif
```

If this file is included and the cpp macro is defined the special directive tells to load trapfpe() and execute it. Thus also feenableexcept() will be executed as well.

C syntax

If you prefer to use the C-syntax:

```
#ifdef FPE_ABORT
#include <cfenv>
 static void __attribute__ ((constructor))
   trapfpe (){
   std::feenableexcept (
         FE_INVALID|FE_DIVBYZERO|FE_OVERFLOW);}};
#endif
```

However the use of static functions is now deprecated!. A

complete example in main_fpe.

Comparing floating points

Beware about comparisons among floating point values:

```
  double a,b;
...
 if (a==b){... // Dangerous!
```

if a and b are the result of arithmetic operations the rounding error may make this comparison meaningless. A better solution:

```
#include<limits>
#include <cmath>
...
double eps=numeric_limits<double>::epsilon();
double tol=eps*max(abs(a),abs(b));
if (abs(a-b)<=tol){...
```

Comparing doubles

Are you sure that `a<b` implies `!(b<a)`?. Indeed the IEEE standard impose that $<$ and $>$ should define good ordering relations among floating point numbers and that this should be guaranteed.

However not all processors are IEEE compliant!. The usual ones implement IEEE standard, don't worry, but if you want to make sure use the compiler option (gnu compiler/clang) −mieee−fp.