Object-Oriented Programming for Scientific Computing

Error Handling and Exceptions

Ole Klein

Interdisciplinary Center for Scientific Computing
Heidelberg University
ole.klein@iwr.uni-heidelberg.de

Summer Semester 2018

Error Handling

If an error occurs in a function of a program, there are several possible treatments (and combinations thereof).

The function could:

1. produce an error message.
2. try to continue.
3. report the error using a return value or a global variable.
4. ask the user for help.
5. shut down the program.

- Combinations of variants (1) to (3) can lead to unpredictable behavior.
- Variant (4) can't be used in non-interactive programs.
- Variant (5) is impossible in vital systems (e.g. aircraft control).

Problem

Often a function cannot decide what to do if an error occurs because not all information necessary for an appropriate response is available locally.

Example 1

- A simulation program asks the user for the number of grid points in x, y and z direction.
- The main program initializes a solver object (e.g. Newton), which in turn creates a linear equation solver, which requires a matrix. There is not enough memory available to store the matrix.
- Now the user should be prompted by the main program to choose a smaller grid size. Within the linear solver, this cannot be done.

Problem

Often a function cannot decide what to do if an error occurs because not all information necessary for an appropriate response is available locally.

Example 2

- During a transport simulation, the linear solver within a Newton iteration does not converge.
- There are several ways to deal with this. One can:
  1. Try using another (possibly computationally more complex) linear solver.
  2. Continue the computation with the currently reached convergence in the Newton method.
  3. reduce the time step size and recompute the time step.
  4. cancel the simulation.
- A choice between these alternatives can only be made at another level of the simulation program (e.g. Newton method, time step control). None of them can be applied locally in the linear solver.

Exceptions

The concept of exceptions can help in these situations:

- Exceptions can transfer the program control across multiple call levels.
- The calling program part decides whether it wants to / can take the responsibility for the solution of a problem in the part that was called.
- Objects of any type can be transferred to the calling level (that may, for example, contain detailed information about the issue).

With exceptions, the error handling is divided into two parts:

1. Reporting an error which cannot be fixed locally.
2. Correction of errors that have occurred in subroutines.

Triggering Exceptions

- If an error occurs, an exception is thrown. To achieve this, an object of arbitrary type is created with the statement `throw`.
- The runtime environment then examines the calling functions one by one, from one level to the other along the call stack, and looks for a part of the program which assumes responsibility for exceptions of this type.
- All local variables in the functions called from that location are destroyed. For variables that are objects the destructor is called.

```
Matrix& Matrix::operator+=(const Matrix& x)
{
    if ( x.numRows != numRows || x.numCols != numCols)
        throw std::string("incompatible dimensions of matrices");
    for (int i = 0; i < numRows; ++i)
        for (int j = 0; j < numCols; ++j)
            entries[i][j] += x.entries[i][j];
    return *this;
}
```

Handling Exceptions

- If a function is willing / able to handle exceptions from certain subroutines, then it announces that by enclosing the corresponding program part in a `try` block.
- Directly after this block one or more `catch` blocks specify which exceptions can be treated and what the reaction should be.

```
Matrix A(4,4,1.), B(3,3,2.);
try
{
    A += B;
}
catch (std::string error)
{
    if (error == "incompatible dimensions of the matrices")
    {
        // something to correct the error
    }
    else
        throw;  // pass on the exception
}
```

Catch Block

- A `catch` block will be executed when
  - one of the expressions in the `try` block has reached a `throw` statement.
  - the `throw` throws an object of the right type.
- In case the object itself is not used in the `catch` block, it is enough to specify its type without a name.
- If a caught exception cannot be handled, or is not completely resolved, it can be thrown further using `throw;`.

Throw

- `throw` creates a temporary object.
- The call stack is searched backwards for the first matching `catch`.
- If there are none available, the program is terminated by calling the function `std::terminate()`. This causes an error message to appear, which gives the type of the object of the exception, e.g.
  `terminate called after throwing an instance of 'std::string'`
- If the program is terminated that way (without exception handler), then it is not guaranteed that the destructors of the objects are called (this is left to the implementation). This can be problematic if e.g. files should be closed in the destructors.

Declaration of Exceptions (Deprecated)

Before C++11, the following was true:

```
Matrix& operator+=(const Matrix& x) throw(std::string);
```

- When declaring a function, one could specify what types of exceptions it might throw (as a comma separated list).
- This helped programmers ensure that all possible exceptions were being treated.
- If another exception was thrown inside the function, then `std::unexpected()` was called, which by default called `std::terminate()`, which in turn called `std::abort()`. `std::unexpected` could be replaced by a user-defined version using the function `set_unexpected`.
- The exception specification had to be repeated for all function declarations and function definitions.
- If the parentheses behind `throw` were empty, no exceptions may be thrown by the function.

C++11: noexcept

The declaration of exceptions via `throw()` is considered deprecated in C++11 and shouldn't be used any more. It is removed in C++17.

There are several reasons for this:

- Exception specifications *are only checked at runtime*, so there is no guarantee that no other exceptions occur.
- Exception specifications *slow down the program*, because there have to be checks for unexpected exceptions.
- If the function throws an unexpected exception, the program is terminated in a suboptimal and unexpected way that is hard to control.
- Exception specifications *do not make sense for template functions*, because the exceptions that could potentially be thrown by e.g. the constructor of a type are unknown at the location of the template.

C++11: noexcept

- In C++11 and above one specifies if a function does *not* throw exceptions. This is done using the keyword `noexcept`.
- If a function defined as `noexcept` throws an exception anyway, the result is *always* a call of `std::terminate()`. This way, there is no additional cost at runtime.
- There are two variants of `noexcept`, a conditional one and an unconditional one, and additionally an operator of the same name.
- The unconditional variant consists of the keyword `noexcept` placed behind the function head:

```
Matrix& operator+=(const Matrix& x) noexcept;
```

C++11: noexcept

- The new operator `noexcept()` returns `false`, if the expression within the parentheses could potentially throw an exception, otherwise `true`.
- This can, for example, be used for optimization, e.g. `std::vector` only uses move semantics if the move constructor of the element type is `noexcept` and creates copies if otherwise.
- In addition, the operator can be used for the conditional variant of the `noexcept` specifier, specifically intended for use with templates. Here a condition is placed in the parentheses that e.g. requires that the effects of certain operators may not cause exceptions to be thrown.

C++11: noexcept

```
#include <iostream>

template<class T>
  T add(T a, T b) noexcept( noexcept(T(a+b)) )
{
  return a + b;
}

int main()
{
  int a = 1, b = 1;
  if (noexcept(add(a,b)))
    std::cout << "exception safe, result is: "
      << add(a,b) << std::endl;
  else
    std::cout << "not exception safe" << std::endl;
  return 0;
}
```

Grouping of Exceptions

Inheritance can be used to create sets of related exceptions:

```
class MathErr {};
class Underflow : public MathErr {};
class Overflow : public MathErr {};
class DivisionByZero : public MathErr {};

void g()
{
    try
    {
        f();
    }
    catch (Overflow)
    {
        // treat all overflow errors here
    }
    catch (MathErr)
    {
        // treat all other math errors here
    }
}
```

All exceptions thrown by the standard library are derived from `std::exception`.

Multiple Inheritance

The occurrence of an error with several consequences or reasons can be expressed through multiple inheritance:

```
class NetworkFileError
    : public NetworkError, public FileSystemError
{};
```

This describes a failure that occurs when accessing an open file on a network, which is both a network error and an error that occurred while accessing the file system.

Together with Design by Contract, this is one of the few areas where use of multiple inheritance is appropriate.

Catching all Exceptions

The expression `catch(...)` catches all exceptions, but it doesn't allow access to the contents of the object. This can be used to clean up locally before the exception is thrown further:

```
try
{
    f();
}
catch (...)
{
    // clean up
    throw;
}
```

Note: If several catch blocks follow one another, they have to be sorted from the most specific to the most general.

Exceptions in Memory Management

- A common reason for throwing exceptions is that more memory has been requested than is available.
- If `new` doesn't get enough memory from the operating system it will first try to call the function `new_handler()`, which can be defined by the user. This could try to free some allocated memory.

```
#include <iostream>
#include <cstdlib>

void noMoreMemory()
{
    std::cerr << "unable to allocate enough memory" << std::endl;
    std::abort();
}  // not a good solution! new_handler is called several times
   // by new and should try to free memory (garbage collection)

int main()
{
    std::set_new_handler(noMoreMemory);
    int* big = new int[100000000000000];
}
```

Exceptions in Memory Management

- If `new_handler()` isn't defined, then the exception `std::bad_alloc` will be thrown.

```
#include <new>

int main()
{
    int* values;
    try
    {
        values = new int[100000000000000];
    }
    catch (std::bad_alloc)
    {
        // do something
    }
}
```

Multiple Resource Allocation

Often, especially in constructors, resources must be allocated several times in succession (opening files, allocating memory, entering a lock in multithreading):

```
void acquire()
{
    // acquire resource r1
    ...
    // acquire resource r2
    ...
    // acquire resource rn
    ...
    // use r1...rn
    // release in reverse
    // release resource rn
    ...
    // release resource r1
    ...
}
```

Problem

- If `acquire rk` fails, `r1,...,r(k-1)` have to be released before cancellation is possible, otherwise a resource leak is created.
- What should be done if allocating the resource throws an exception that is caught outside? What happens to `r1,...,r(k-1)`?
- Variant:

```
class X
{
  public:
    X();
    ~X();

  private:
    A* pointerA;
    B* pointerB;
    C* pointerC;
};
```

```
X::X()
{
    pointerA = new A;
    pointerB = new B;
    pointerC = new C;
}
```

Solution

"Resource Acquisition is Initialization" (RAII):

- is a technique which solves the problem above.
- is based on the properties of constructors and destructors and their interaction with exception handling.
- is actually a misnomer: "Destruction is Resource Release" (DIRR) would be more appropriate, but the acronym RAII is now too well-known to change it.

RAII is a specific way of thinking about resources that originated in C++ and provides an elegant alternative to strategies used in Java, etc. (and has a really unfortunate name for something so central...).

Rules for Constructors and Destructors

Central rules that enable RAII:

1. An object is only fully constructed when its constructor is finished.
2. A compliant constructor tries to leave the system in a state with as few changes as possible if it can't be completed successfully.
3. If an object consists of sub-objects, then it is constructed as far as its parts are constructed.
4. If a scope (block, function...) is left, then the destructors of all successfully constructed objects are called.
5. An exception causes the program flow to exit all blocks between the `throw` and the corresponding `catch`.

Implementation

```
class A_ptr
{
  public:
    A_ptr()
    {
      pointerA = new A;
    }
    ~A_ptr()
    {
      delete pointerA;
    }
    A* operator->()
    {
        return pointerA;
    }

  private:
    A* pointerA;
};

// corresponding classes
// B_ptr and C_ptr
```

```
class X
{
  // no constructor and destructor
  // needed, the default variant
  // is sufficient
  private:
    A_ptr pointerA;
    B_ptr pointerB;
    C_ptr pointerC;
};

int main()
{
    try
    {
        X x;
    }
    catch (std::bad_alloc)
    {
        ...
    }
}
```

(This is actually a simple mock-up of smart pointers)

Implementation

Basic principle:

- The constructor `X()` calls the constructors of `pointerA`, `pointerB` and `pointerC`.
- When an exception is thrown by the constructor of `pointerC`, then the destructors of `pointerA` and `pointerB` are called and the code in the `catch` block will be executed.
- This can be implemented in a similar fashion for the allocation of other resources (e.g. open files).

Main idea of RAII:

- Tie resources (e.g. on the heap) to handles (on the stack), and let the scoping rules handle safe acquisition and release
- Repeat this recursively for resources of resources
- Let the special rules for exceptions and destructors handle partially-constructed objects

Implementation with C++11

```
#include <memory>

class A {};
class B {};
class C {};

class X
{
  public:
    X() : pointerA(new A),
          pointerB(new B),
          pointerC(new C)
    {}
    // no destructor necessary,
    // default variant is sufficient
  private:
    std::unique_ptr<A> pointerA;
    std::unique_ptr<B> pointerB;
    std::unique_ptr<C> pointerC;
};
```

```
int main()
{
    try
    {
        X x;
    }
    catch (std::bad_alloc)
    {
      ...
    }
}
```

Key Assumptions for Exception Handling in C++

Key assumptions for exceptions:

1. Exceptions are mainly used for error handling.
2. There are few exception handlers compared to function definitions.
3. Thrown exceptions are rare in comparison with function calls.
4. Exceptions are part of the language, not just a convention for error handling.

Consequences:

- Exceptions are not just an alternative to the return mechanism, but a mechanism for the construction of fault tolerant systems.
- Not every feature must be a fault tolerant unit. Instead, whole subsystems may be fault tolerant, without every function having to implement this functionality.
- Exceptions should not be the sole mechanism for error handling, only an extension for cases that can not be solved locally.

Ideals for Exception Handling in C++

Features exception handling should have:

1. Type-safe transfer of any information from the throw point to the handlers.
2. Don't cause costs (at runtime or in memory) if no exception is thrown.
3. Guarantee that any exception is caught by an appropriate handler.
4. Allow grouping of exceptions.
5. The mechanism is supposed to work in multi-threaded programs.
6. Cooperation with other languages (especially C) should be possible.
7. Easy to use.
8. Easy to implement.

(3) and (8) were later considered too costly or too restrictive and are only achieved to some extent.

The term `throw` was chosen because `raise` and `signal` were already assigned to C library functions.

Resumption or Termination

During the design of exceptions it was discussed whether the semantics of exceptions should be terminating or resuming. Resumption means that the routine continues after the exception has been handled, e.g.:

- a handler routine is started in case of lack of memory, finds new memory and then returns to the point of the call
- a routine is started because the CD drive is empty, it then asks the user to insert the CD and returns

The main reasons for resumption:

- Recovery is a more general mechanism than termination.
- In the case of blocked resources (CD is missing,...) resumption provides an elegant solution.

Resumption or Termination

The main reasons for termination:

- Termination is significantly easier.
- The treatment of scarce / missing resources with recovery leads to programs that are error prone and hard to understand because of the close coupling between libraries and their users (consumers).
- Large software systems have been written without resuming, therefore it is not absolutely necessary. This can e.g. be seen in the case of Xerox Cedar/Mesa, a fossilized programming language that should support resumption. It had $\sim$ 500.000 lines of code, but resumption existed in only one (!) place, all other uses of resumption gradually had to be replaced by termination. This place was a context inquiry, where resumption was unnecessary.

$\Rightarrow$ **Therefore termination is the standard in C++**

C++11: Exception Pointer

- In C++11 a special kind of pointer, `std::exception_ptr`, was introduced, which can store exceptions and also pass them on for subsequent treatment. It can accommodate *all* types of exceptions.
- With the function `std::current_exception()` a pointer to the currently thrown exception can be obtained in a `catch` block. Alternatively, a `std::exception_ptr` can be generated from an exception object in a `catch` block using the function `std::make_exception_ptr`.
- The function `std::rethrow_exception`, which expects a `std::exception_ptr` as argument, can be used to throw the exception again and then treat it.
- Exception pointers are especially relevant for multithreading, because they can be used to communicate exceptions (later).

Example

```cpp
#include <iostream>
#include <exception>

void treatEptr(std::exception_ptr exPtr) // passing Eptr
{
    try {
        if (exPtr != nullptr) {
            std::rethrow_exception(exPtr);
        }
    } catch(const std::string& e) {       // treating exception
        std::cout << "Exception \"" << e << "\" caught" << std::endl;
    }
}

int main()
{
    std::exception_ptr exPtr;
    try {
        throw(std::string("foo"));
    } catch(...) {
        exPtr = std::current_exception(); // capture
    }
    treatEptr(exPtr);
} // destructor for stored exception is called here!
```

Exceptions and Assertions

- The language device of assertions already existed in C, with a macro `assert` whose argument is a comparison. If the comparison is false, the program will give an error message of the type:

  `Assertion failed: expression, file filename, line number`

  and exit.
- The macro is defined in the header file `assert.h` (or equivalently `cassert`).
- If the variable `NDEBUG` is set when compiling a program (either through a *`#define NDEBUG`* in a source file or a `-DNDEBUG` in the compiler call), all assertions will be ignored.
- The main purpose of assertions is catching programming errors, they are usually disabled in the final version of a program for performance reasons.
- By contrast, exceptions are used to handle errors during a normal program run, particularly those which can be resolved automatically.

Example

Program:

```
#include <assert.h>
#include <iostream>

int divide(int a, int b)
{
  assert(b != 0);
  return (a/b);
}

int main()
{
  int a = 1, b = 0;
  std::cout << "The quotient is: " << divide(a,b) << std::endl;
}
```

Output:

```
<file and line>: int divide(int, int): Assertion 'b != 0' failed.
Aborted
```

Example

Program:

```
#include <assert.h>
#include <iostream>

int divide(int a, int b)
{
  assert(b != 0);
  return (a/b);
}

int main()
{
  int a = 1, b = 0;
  std::cout << "The quotient is: " << divide(a,b) << std::endl;
}
```

Output translated with `-DNDEBUG`:

```
Floating point exception
```

Summary

- Exceptions can be used to handle runtime problems that can't be treated locally (not enough memory, solver breakdown,...)
- Functions that never throw exceptions can be marked `noexcept`, e.g. for optimizations in container classes
- RAII is a central concept of modern C++ that makes resource handling much safer, based on the interplay of exceptions and destructors
- Exceptions in C++ terminate the caller, because resumption proved too difficult and largely unnecessary
- C-style `assert` is for compile time bug detection, and should not be confused with exceptions for runtime error detection