Exception handling

how to deal with errors at run time

- return 0 or 1 or infinity?
- ignore?
- abort?
- return error flag?

none of these is ideal

- when returning 0 or 1 how to make sure the program continues meaningfully?
- error flags are easily ignored by users
- abort : should unexpected errors lead to a loss of the whole calculation?

C++ offers the exception class as a solution

- the library encounters an exception but does not know how to deal with it
- it throws an exception
- the calling program might be able to deal with an exception
- the calling program catches the exception
- uncaught exceptions lead to the *termination* of the program
- does not help for segmentation faults or other (dumb) pointer-based errors

try…throw…catch…

The syntax for error handling is the try…throw…catch… block

```
#include <iostream>
using namespace std;

int main () {
  try
  {
    throw 20;           // error nr 20
  }
  catch (int e)
  {
    cout << "An exception occurred. Exception Nr. " << e << '\n';
  }
  return 0;
}
```

Errors of different type can be caught as follows:

```
try {
  // comment any of the lines below and compare the output
  throw 'a';
  throw 5;
  throw 5.0;
  cout << "This line will not be executed\n";
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

try…throw…catch…

Nesting is also allowed:

```
try {
  try {
    throw 20;
  }
  catch (int n) {
    throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

throws the exception again

The inner catch block can throw an exception which is then caught by the outer catch block

Throwing an exception interrupts the normal execution

- the stack is unwound, functions are exited, local objects are destroyed
- only objects that are fully constructed are destroyed, otherwise its destructor is not called
- until a catch clause is found

exception class

The C++ exception class looks like this:

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

C++11

the noexcept at the end guarantees that these functions never throw exceptions (in C++98 this is throw().). Let us look at an example how to use it:

standard exception class

it contains a virtual member function what() that can be overwritten in derived classes to provide information on the type of error. It returns a const char*

```
// using standard exceptions        -- C++11
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
  virtual const char* what() const noexcept
  {
    return "My exception happened";
  }
} myex;

int main () {
  try
  {
    throw myex;
  }
  catch (exception& e)
  {
    cout << e.what() << '\n';
  }
  return 0;
}
```

Through the ampersand (&) we also catch exceptions of types derived from exception such as myexception

standard exception

Let us see how inheritance, polymorphism, and slicing (when invoking the copy constructor) are useful for exception handling:

C++11

```
// exception constructor
#include <iostream>       // std::cout
#include <exception>      // std::exception

struct ooops : std::exception {
  const char* what() const noexcept {return "Ooops!\n";}
};

int main () {
  ooops e;
  std::exception* p = &e;
  try {
      throw e;       // throwing copy-constructs: ooops(e)
  } catch (std::exception& ex) {
      std::cout << ex.what();
  }
  try {
      throw *p;      // throwing copy-constructs: std::exception(*p)
  } catch (std::exception& ex) {
      std::cout << ex.what();
  }
  return 0;
}
```

What is the output of this program?

standard exception

The standard library provides a base class std::exception defined in the header <exception>. It has several derived types http://www.cplusplus.com/reference/exception/exception/

- bad_alloc : thrown by new on allocation failure
- bad_cast : thrown by dynamic_cast (see ex13.cpp)
- bad_exception : thrown by certain dynamic exception specifiers
- bad_typeid : thrown by typeid
- bad_function_call : thrown by empty function objects (C++11)
- bad_weak_ptr : thrown by shared_ptr when passed a bad weak_ptr (C++11)
- logic_error : an error related to the internal logic of the program (domain_error, out_of_range, length_error are members)
- runtime_error : errors detected during runtime (overflow_error, underflow_error, and range_error are members)

The last two can be inherited by custom exceptions to report errors:

```
// bad_alloc standard exception
#include <iostream>
#include <exception>
using namespace std;

int main () {
  try
  {
    int* myarray= new int[1000];
  }
  catch (exception& e)
  {
    cout << "Standard exception: " << e.what() << endl;
  }
  return 0;
}
```

example: domain_error

example: the square root of a function is only defined for non-negative numbers. Thus, a negative number for such a function would qualify as a domain error

```
class domain_error : public logic_error {
public:
  explicit domain_error (const string& what_arg);
  explicit domain_error (const char* what_arg);
};
```

constructor: the value as what_arg has the same content as the value returned by member what
The class inherits the what member from logic_error

Exception safety:
strong guarantee: if the constructor throws an exception, there are no side effects

http://www.cplusplus.com/reference/stdexcept/domain_error/

standard exception

- bool std::uncaught_exception() detects if the current thread has a live exception object, ie if stack unwinding is in progress
- std::terminate() is called when exception handling fails. The program terminates
- std::unexpected() is called when an exception is thrown from a function whose exception specification forbids exceptions of this type; will most likely result in std::terminate()

Dynamic exception specification:

- void f() throw(int); // guarantees to throw only an int; otherwise will result in std::unexpected() — C++98
- void f() throw; // guarantees to throw no exceptions, all errors result in std::unexpected() — C++98
- void f(); // normal exception handling
- as of C++11: void f() noexcept; and void f() noexcept(false);

implications

One has to assume that exceptions can occur at any time in the code. As an example let us look at the following code. Is it safe?

```
void SimpleFunction() {
  int* myArray = new int[1000];
  Some_operation(myArray);
  delete [] myArray;
}
```

In case Some_operation throws an exception we have a memory leak

- do not use exception handling (eg, Mozilla firefox web browser)
- catch-and-rethrow
- object memory management

```
void SimpleFunction() {
  int* myArray = new int[1000];
  try {
    Some_operation(myArray);
  }
  catch (...) {
    delete [] myArray;
    throw;
  }
  delete [] myArray;
}
```

This is what the catch-and-rethrow approach might look like. The lone throw rethrows the caught exception. We need to be inside the catch clause of Some_operation, otherwise the program will crash. We have to duplicate the delete [] which looks like bad style

smart pointers

C++11

How to make sure that we clean up all memory?
Answer: smart pointers, a kind of encapsulated pointers that properly clean up the resource when destroyed. There exist different types: shared_ptr, weak_ptr, unique_ptr etc. The concept of smart pointers existed long before C++11:
in the header file <memory> there is the templated auto_ptr. It accepts in its constructor a pointer to dynamically allocated memory and calls delete (not delete [], so you cannot store dynamically allocated arrays in auto_ptr). Through operator overloading you can use dereferencing and arrow operations as though it were an ordinary pointer. However, the syntax of auto_ptr was essentially broken. auto_ptr should never be used. We will focus here on unique_ptr.

```
unique_ptr<vector<int> > pVector(new vector<int>);
pVector->push_back(100); // Add 100
(*pVector)[0] = 200; // Dereferencing works as usual
```

A unique pointer differs fundamentally in assignment and initialization from ordinary pointers

```
unique_ptr<int> one(new int);
*one = 1;
unique_ptr<int> two;
two = std::move(one);
cout << "one : " << (one == nullptr ? "nullptr" : "1") << " two : " << *two << "\n";
```

After these lines, *two* contains what *one* originally contained whereas one is nullptr. Using *one* from now on would lead to trouble

smart pointers

- there can be at most one unique_ptr to a resource
- you don't have to worry about clean-up
- it is safe to return unique_ptr from functions without clean-up (see below and ex9.cpp)
- corresponds to *move semantics* (see chapter on C++11)

member functions:

- (constructor)
- (destructor)
- get()
- operator* (dereferencing)
- operator->
- reset
- release
- swap
- operator=(T&& rhs)

http://www.cplusplus.com/reference/memory/unique_ptr/

example : see ex6.cpp

having objects manage resources through their constructors and destructors is commonly referred to as *resource acquisition is initialization* (RAII)

smart pointers and exception handling

Smart pointers are invaluable in combination with exception handling. Consider

```
class mylist {
public:
  int element;
  mylist *next;
};
```

```
mylist m;
mylist* m2 = GetNewList();
cout << m2->element << "\n";
```

In the code below, a memory leak would occur if Somecalculation() produces an exception:

```
mylist* GetNewList() {
  mylist* newlist = new mylist;
  newlist->next = nullptr;
  newlist->element = SomeCalculation();
  return newlist;
}
```

```
delete m2;
```

By using a smart pointer this problem can be solved because local variables are unwound from the stack when an exception is caught:

```
mylist* GetNewList_better() {
  unique_ptr<mylist> newlist(new mylist);
  newlist->next = nullptr;
  newlist->element = SomeCalculation();
  return newlist.release();
}
```

example based on http://web.stanford.edu/class/cs106l/course-reader/Ch12_ErrorHandling.pdf

smart pointers and exception handling

However, when calling GetNewList_better() without left-hand-side a memory leak would still occur because the caller does not clean up the memory. But this problem can also be eliminated:

```
unique_ptr<mylist> GetNewList_evenbetter() {
  unique_ptr<mylist> newlist(new mylist);
  newlist->next = nullptr;
  newlist->element = SomeCalculation();
  return newlist;
}
```

For *runtime logical errors* we recall assert defined in <assert.h>. It is better suited for coding bugs. It slows code down, but most compilers disable it in release or optimized builds. Example:

```
int operation_on_even_int(const int n) {
  assert(!(n%2));
  return n/2;
}
```

Invariants can (should) always be checked like this.

exceptions vs function call

Although at first it may seem that throw catch blocks and function calls behave in a similar way, there are notable differences:

- in both cases pass by value, pointer and reference are legal syntax; however, as we will see below, exceptions should be passed by reference only
- after a function call control is given back to the caller; after a catch this is not the case
- *an exception must always be copied* (even if called by reference or if a static variable is called by reference): *a local object is needed*, it is not possible for a catch clause to modify the object.
- if a catch clause is encountered for a parent class but called with an argument from a derived class, it will be evaluated (cf runtime_error and range_error). In case a range_error is thrown and a catch clause for a runtime_error is provided, the catch clause will be evaluated
- catch clauses are evaluated in the order of appearance; ie if a catch specification for a derived class appears after the one for a parent class, it will never be evaluated.

Constructors

Prevent resource leaks in constructors. If a class constructor calls multiple constructors of dynamic data (via new) and an exception occurs during one of them, then the class destructor is not called, resulting in a resource leak.

Safe syntax looks like this:

```
class Simulation {
public:
    Simulation(const string&, const Model&, const Lattice&);  // constructor
    /* more public member functions */
private:
    std::string MyName;
    unique_ptr<Model> MyModel; // model
    unique_ptr<Lattice> MyLattice; // lattice
};

Simulation::Simulation(const string& name, const Model& m, const Lattice& latt) : MyName(name), MyModel(new Model(m)), MyLattice(new Lattice(latt)) {}
```

If an exception occurs during initialization of MyLattice, MyModel is already a fully constructed object and will be destroyed, just like MyName. In addition, they are full objects and will automatically be destroyed when the Simulation object containing them is. So nothing needs to be written in the destructor of Simulation.

Compare this to the same code where bare pointers are used.

Destructors

Prevent resource leaks in destructors. Keep exceptions from propagating out of destructors:

- if an exception occurs before completion of a destructor not all resources are released
- it prevents calling of std::terminate

```
Matrix::~Matrix() {
  try {
    logDestruction(this);
  }
  catch (...) {}
  /* more code */
}
```

Imagine an exception occurs in logDestruction. Without try…catch the error propagates beyond the destructor and terminate will be called. More code is not executed. This can be improved with the try…catch block.

In the example above the catch block prevents here the propagation beyond the destructor, so it is not doing nothing even with just {}!

Pass exceptions by reference, throw by value

Do not pass exceptions by pointers:

- since the exception must be copied, a (local) pointer will go out of scope before execution and this *makes* hence *no sense*. Static and global pointers could work
- an object op the heap (with new) leads to ambiguities that can not be answered: who should delete the resource on the heap?

Do not pass exceptions by value:

- pass by value requires two copies: the exception and the copying of it into the argument
- since copying is based on the *static* type, the expected result of the exception may differ from the intended one in case of class derivations (cf the example before on slide nr 7 on slicing)

```
DerivedClassObject d;
ParentClassObject& p = d;     // p refers to DerivedClassObject
throw p;                      // this throws an exception of type ParentClassObject
```

Always pass exceptions by reference:

- only one copy operation (cf ex7.cpp) — you always need a local object!!
- avoids the problems of temporaries
- fixes the problem with inheritance

why exception handling is slow

- exception handling should be used for a major change in the flow of control only
- while you can use exception handling as a fancy form of function call and return, do not do so (see also before) : the cost of throwing an exception is about 1000 times bigger than a normal function return
- if any part of a program (including a library) uses exceptions, the rest of the program must support them, too. So you cannot avoid it completely
- throwing an exception is much slower than returning a function value because of the copying involved
- bookkeeping of all types that require destruction at any moment of the runtime is required (assuming an error can occur at any time)
- whenever a try block is entered, exceptions for all possible types in the catch clause must be considered (extra bookkeeping)
- runtime comparisons are needed to ensure that exception specifications are satisfied
- this leads to slower code as well as larger files (both around 5-10%) : avoid hence unnecessary try blocks in performance-critical regions of the program.