Software Design (C++)

3. Resource management and exception safety (idioms and technicalities)

Juha Vihavainen
University of Helsinki

Preview

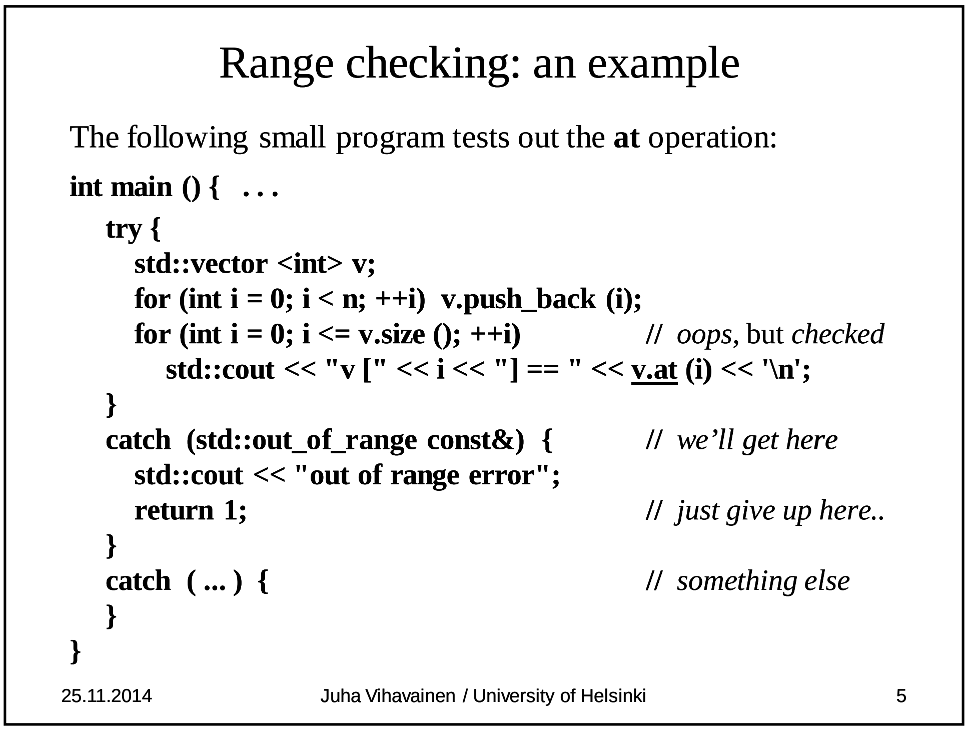
- More on error handling and exceptions
  - checking array indices (again) - why not?
  - handling complicated error situations
  - the RAII principle
- *Smart pointers* and resource management
- Implementing *exception safety*
  - built-in language support
  - programming techniques and idioms

Index checking

```
// an almost real vector of doubles:
# include <stdexcept>   // . . .          // get std::out_of_range
class Vector {                             // my custom container. . .
public:   // . . .
    double& operator [] (int n);           // unsafe but max efficiency
    double& at (int n);                    // index is checked
    // . . .
};
// possible implementations (as allowed by the C++ standard)
double& Vector::operator [] (int n)  {  return elem [n]; }
double& Vector::at (int n)  {
    if (n < 0 || sz <= n) throw std::out_of_range ("at: invalid index");
    return elem [n];
}
```

Reminder: Why not index checking?

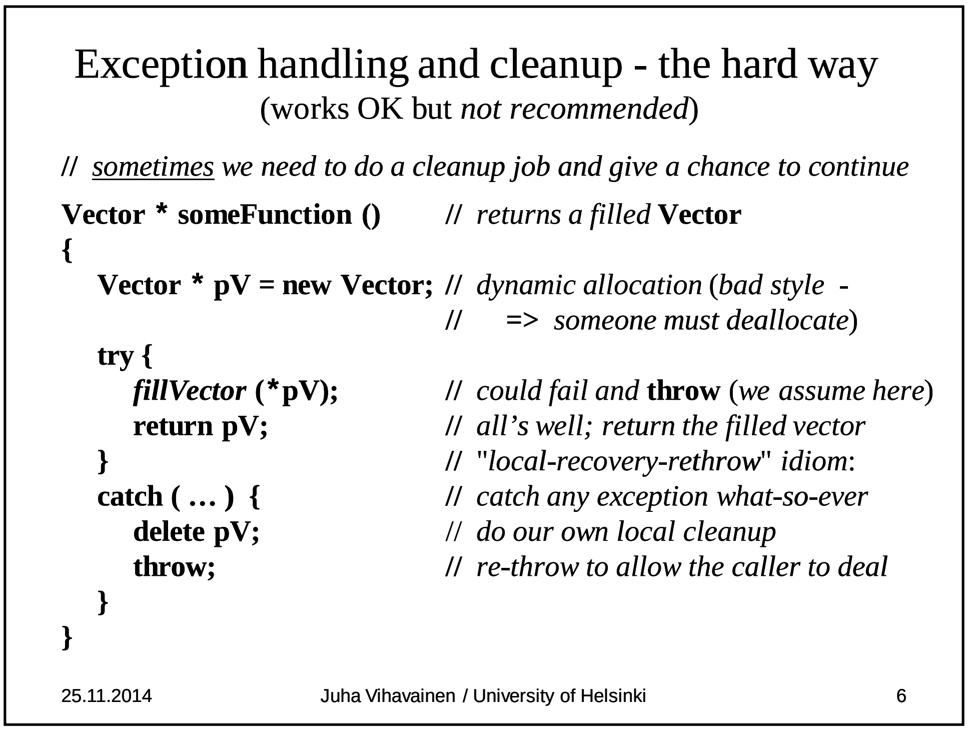
- Checking costs in speed and code size
  - not much, don't worry
  - few real-world projects need to worry
- But *some* projects may really require optimal performance
  - think huge (e.g., Google) and very tiny (e.g., cell phone)
- The standard must serve everybody (is still competing with C)
  - "*you can't build optimal on top of checked*"
  - "*you can build checked on top of optimal*"
- Some projects cannot even be allowed to use exceptions
  - old projects with pre-exception parts
  - high reliability, hard real-time code (think airplanes)
- The C part of C++ can't give security guarantees anyway!



Range checking: an example

The following small program tests out the **at** operation:

```
int main () {   . . .
    try {
        std::vector <int> v;
        for (int i = 0; i < n; ++i)  v.push_back (i);
        for (int i = 0; i <= v.size (); ++i)              //  oops, but checked
            std::cout << "v [" << i << "] == " << v.at (i) << '\n';
    }
    catch  (std::out_of_range const&)  {               //  we'll get here
        std::cout << "out of range error";
        return 1;                                      //  just give up here..
    }
    catch  ( ... )  {                                  //  something else
    }
}
```



Exception handling and cleanup - the hard way

(works OK but *not recommended*)

```
//  sometimes we need to do a cleanup job and give a chance to continue
Vector * someFunction ()        //  returns a filled Vector
{
    Vector * pV = new Vector;   //  dynamic allocation (bad style  -
                                //      =>  someone must deallocate)
    try {
        fillVector (*pV);       //  could fail and throw (we assume here)
        return pV;              //  all's well; return the filled vector
    }                           //  "local-recovery-rethrow" idiom:
    catch ( … )  {              //  catch any exception what-so-ever
        delete pV;              //  do our own local cleanup
        throw;                  //  re-throw to allow the caller to deal
    }
}
```

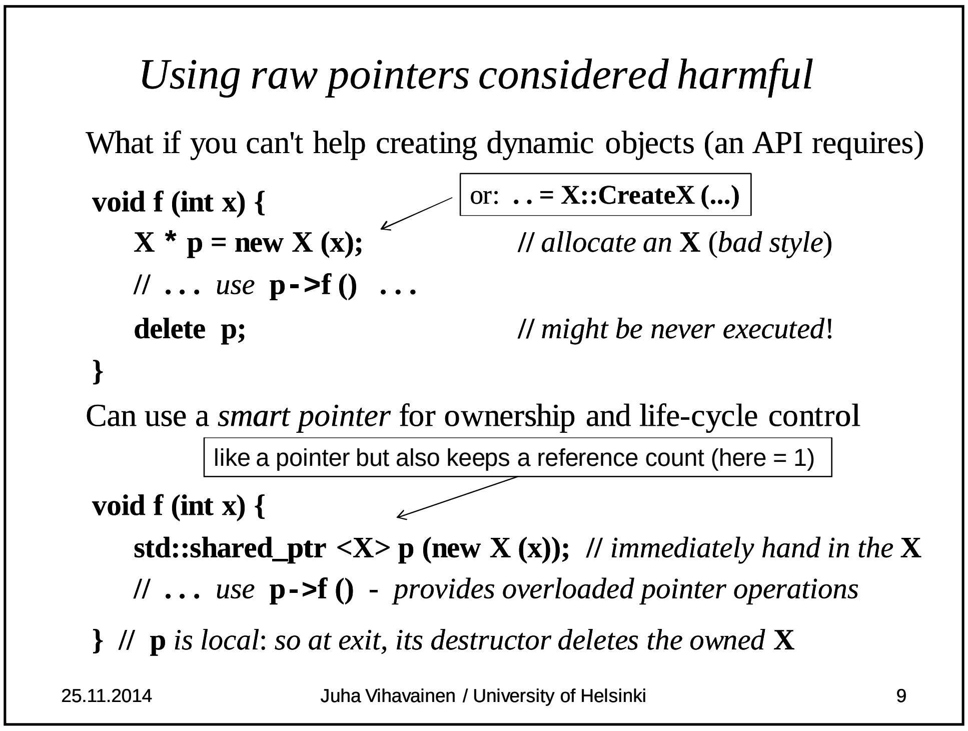
Exception handling with destructors
(simpler and more structured idiom)

```
// using scoped (local) variables makes the cleanup automatic
void someOtherFunction ()      // uses a filled vector
{
    Vector v;                  // vector handles deallocation
    fillVector (v);            // pass as a reference
    // use v here . . .
}                              // v is automatically destructed
```

- the compiler-generated destructor call at end of block works like an implicit **finally** clause
- if you feel that you need a **try-catch** block: think again
  - you might be able to do much better without it

RAII: *Resource Acquisition Is Initialization*

- **std::vector**
  - acquires memory for elements in its constructor (or later)
  - manages it (changing size, controlling access, etc.)
  - gives back (releases) the memory in its destructor
- This is a special case of the general resource management strategy called RAII (for in-depth discussion, see [Stroustrup, 2014, Ch. 19.5])
  - also called “scoped resource management”
  - use it wherever you can
  - simpler and cheaper than anything else
  - interacts elegantly with error handling using exceptions
  - examples of resources: memory, file handles, sockets, I/O bindings (iostreams handle those using RAII), locks, widgets, threads



Using raw pointers considered harmful

What if you can't help creating dynamic objects (an API requires)

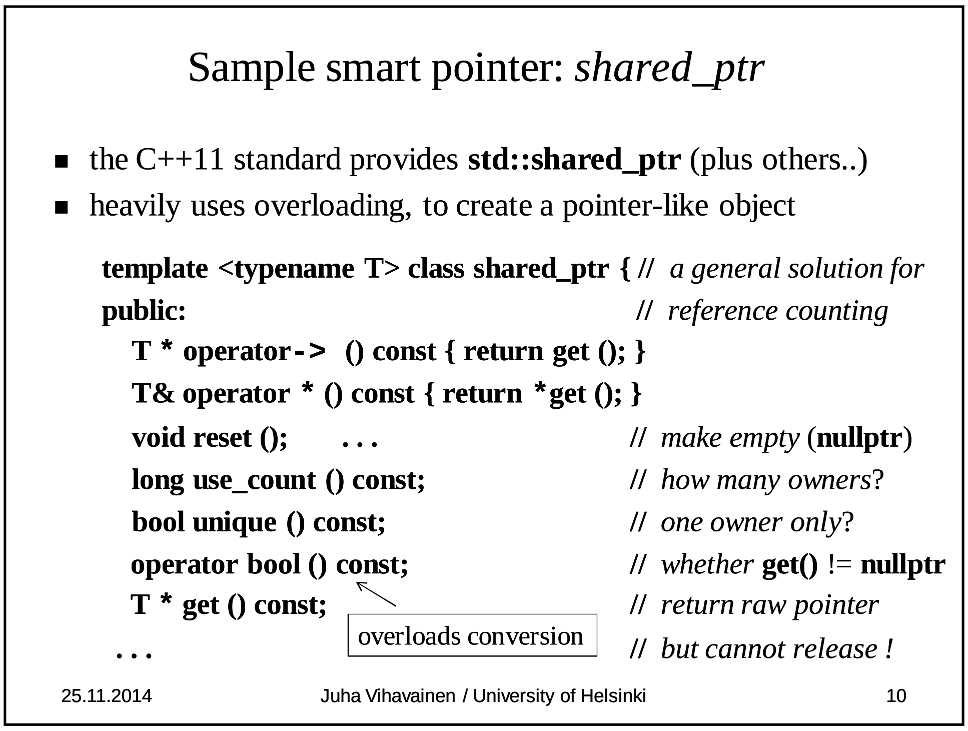
or: **. . = X::CreateX (...)**

```
void f (int x) {
    X * p = new X (x);              // allocate an X (bad style)
    // . . . use p->f () . . .
    delete p;                       // might be never executed!
}
```

Can use a *smart pointer* for ownership and life-cycle control

like a pointer but also keeps a reference count (here = 1)

```
void f (int x) {
    std::shared_ptr <X> p (new X (x));  // immediately hand in the X
    // . . . use p->f () - provides overloaded pointer operations
} // p is local: so at exit, its destructor deletes the owned X
```



Sample smart pointer: *shared_ptr*

- the C++11 standard provides **std::shared_ptr** (plus others..)
- heavily uses overloading, to create a pointer-like object

```
template <typename T> class shared_ptr { //  a general solution for
public:                                  //  reference counting
    T * operator-> () const { return get (); }
    T& operator * () const { return *get (); }
    void reset ();      . . .            //  make empty (nullptr)
    long use_count () const;             //  how many owners?
    bool unique () const;                //  one owner only?
    operator bool () const;              //  whether get() != nullptr
    T * get () const;                    //  return raw pointer
  . . .                                  //  but cannot release !
```

overloads conversion

Safe exception handling: background

- Originally, exceptions and errors were very poorly understood aspect of C++; also, expection safety issues and requirements were seen and added to the C++89/03 standard at last moments

  [Stroustrup, 2000] *App*. E: *Standard-Library Exception Safety*

  Article on *Exception Safety* **http://www.stroustrup.com/except.pdf**

  Boost article: *Exception-Safety in Generic Components*
  http://www.boost.org/community/exception_safety.html

- Two concerns:
  1. class invariants must be maintained - or restored
  2. no resources may be leaked
     - including: memory space, opened files, locks, connections, or any other shared system resources

Invariants revisited

- Use defensive programming and self-checking objects
- A *class invariant* is an assertion that holds before and after any operation manipulating an object
- Preconditions tests "external" failures which the unit cannot handle itself: must throw an exception

```
if (! precondition )
    throw AnException ("diagnostics");   // back to the caller
```

- Invariants and (testable) postconditions check internal states that don't make sense to outsiders, and most often indicate a bug in code => use asserts to eliminate them

```
assert ( isInValidInternalState_);          // aborts if not
```

- Often, we don't know the actual reason of a failure: perhaps a programming error or some external factor

Invariants revisited (cont.)

- The *programming-by-contract* separates responsibilities but actual production software is not so that clear cut and clean
- Preconditions provide a pragmatic trade-off *what to check*, at the boundary of a unit (class/function/method)
- Note that the C++ standard library uses the same strategy (provides checks for selected operations that may throw)

The *fundamental problem* with exception safety

- an exception thrown from some component or function may interrupt the algorithm and leave the state of the calculation (objects) in some bad or indeterminate state
- if the class invariant does not hold, the object cannot be even *destructed* without causing undefined behavior

Many sources of exceptions

- User-supplied and system functions, such as allocator functions, can throw exceptions (from [Stroustrup, 2000])

```
void fun (std::vector <X>& v,  X const& x) {
                                    // sample exceptions:
    v [2] = x;                      // X's assignment may throw
    v.push_back (x);                // vector's allocator may throw
    std::vector <X> u = v;          // X's copy ctor may throw
       . . .
}    //  u is destructed here;  X's dtor should not throw!
```

- Also, IO operations may throw (if so configured)

Levels of exception safety

1. *Basic Guarantee*: no leaks, and maintains class invariant.
2. *Strong Guarantee*: succeeds, or leaves state unchanged.
3. *Nofail Guarantee*: doesn't fail (throw) in any circumstances.

- the last one (*Nofail*) is often needed to implement the former ones; e.g., an assignment of a primitive value (say, a pointer) cannot fail - but anything with allocations may fail!
- the *strong guarantee* for some complicated update may require a "roll-back" mechanism (sometimes too expensive)
- "*maintaining class invariant*" means
  - the object is in *some* valid state (but not necessarily in the one we would ideally like it to be)
  - but it can be at least released and *destructed*

Levels of exception safety_(cont.)

- e.g., **std::vector <T>::push_back** is designed to give the strong guarantee: a new element is added or *no* change

Additionally:

4. *Exception Neutrality*: exceptions originating from components are always passed through unmodified

- relevant for a container handling and copying its elements, especially when using C++ templates
- e.g., standard **vector <T>::push_back** also manifests *exception neutrality*: after any internal clean-up, propagates the original exception caused by the copying operation - that depends on the actual element type (**T**)

User can call constructors and destructor

- Constructors & destructors are *usually* called by compiler

  X * ptr = new X; . . . // *reserve memory, then construct* **X**

  delete ptr; // *destruct* **X**, *then release its memory*

  - note that if **p** is **nullptr** (zero), **delete** has no effect
- When necessary, *allocation* can be separated from object *initialization* with the so-called *placement-new* operator

  void * p = ::operator new (sizeof (X)); // *allocates space*

  ptr = new (p) X; // *placement-new constructs* **X** *at* **p**
- Similarly, we could separate destruction & deallocation

  ptr-> ~X (); // *destructs the object pointed by* **ptr**

  ::operator delete (ptr); // **delete operator** *frees space*
- Since a destructor is a member function, we can call it explicitly - but then must very carefully ensure that the compiler doesn't!

Exceptions and ctors/dtors

The following built-in C++ mechanisms enable resource management even in case of failures and exceptions

1. An exception throw causes the unwinding of call stack
   - all objects located (in the call stack) between the places where the exception is thrown and caught are destroyed, i.e., their destructors are called
2. Suppose that an exception is thrown inside a constructor, which has already constructed one or more embedded members (fields)
   - the run-time system ("compiler") calls the destructors of the already constructed members to release resources reserved by those members
3. A failed construction in a "**new T ()**" operation also always
   - releases the space allocated for the **T** object

Language support for exception safety

C++ language rules/implementation ensure that exceptions thrown while construction will be handled correctly

1. Either the object is *fully built* (and its invariants OK), or its members become automatically destructed (by the compiler/run-time system)
2. Also **new** operations are implemented safely; "**p = new T;**" is *compiled* into something like (pseudocode here):

```
p = allocate space for a T:   // may fail & throw but that's OK!
try {  construct a T at p;    // create it here (placement new)
}  catch  ( ... )  {          // note exception neutrality
    free the reserved space;    // release back raw memory
    rethrow the same exception;
}
```

- the T ctor is assumed to be "safe": no leaks of its own

Case: how safety mechanisms work

- Consider the following C++ class and code

```
class A : public B {
public:
    A () {}                      // implicit ctor calls for x and y
    X x;   Y y;                  // two (public) members..
};  . . .                        // using implicit dtor ~A (), here

A * a = new A;       . . .       //  . . . some other code
delete a;
```

- The **A** constructor *may seem empty* but actually it contains the construction of the **B**, **X**, and **Y** parts of an **A** object
- Similarly **A**'s compiler-generated destructor handles the destruction of all these members

Case (cont.)

```
A * p = new A;     . . .           // create a dynamic A and use it
delete p;                          // later get rid of it
```

- The above code is *implemented* by the *compiler* as follows

```
void * p = ::operator new (sizeof (A));       // (1) allocate
    // operator new throws bad_alloc upon failure (no problem)
try {                 may throw    // we now have space for an A
   new (p) A; }                    // (2) create an A at p (or fail)
catch ( ... ) {                    // handle whatever failures
   ::operator delete (p);  throw;
}
                    cannot
 . . .              throw          // some other code . . .
((A*)p)-> ~A ();                   // (2') release A-specific resources
::operator delete (p);             // (1') release A's space (via p)
```

On implementing your own strong guarantee

- a sketch of strong exception guarantee and *roll-back*

```
void doOperation (T const& value) {
   try {  < update the state copying the giving value >;
                // e.g., a copy operation may fail & throw
   } catch ( ... )  {               // catch any exception
      < restore the old state and its invariants >;
      throw;                        // now rethrow the original
   }
}
```

- exception neutrality: any **T**-related exceptions pass through
- strong guarantee may be very tricky or too costly to achieve; even STL does not provide it for *all* its operations
- special C++ idioms support strong guarantee (see later)

Example: exception safe constructor for *Vector*

STL container operation

```
Vector::Vector (size_t sz, T const& x) {  // create a vector of sz x's
    rep_= (T*)::operator new (sz*sizeof(T));  // new may fail
    T * p = rep_;                              // element address
    try {                                      // construct sz items
       for (; p != rep_+ sz; ++p)  new (p) T (x);   // ctor may fail
    } catch ( ... ) {                 // handle T constructor failures
       while (p-- != rep_)            // destroy all constructed items
          p->T::~T ();                // call T's destructor
       ::operator delete (rep_);      // release memory
       throw;                         // propagate the original exception
    }
    size_= capacity_= sz;               // OK: Vector initialized
}
```

Exception-safe copy ctor, and assignment

- Similar implementation must be written for *copy construction*: if copy of one item fails, must destruct the previously copied ones . .
- *Assignment operators* can often be safely programmed with an existing copy constructor and a (safe) **swap**:

```
Vector& Vector::operator = (Vector const& rhs) {
    Vector tmp (rhs);       // may fail (but no problem)
    // can now do the actual assignment, safely and efficiently
    swap (tmp);             // does not fail (exchanges fields)
    return *this;
}                           // tmp is destructed here (old value)
```

  - we assume that the **swap** operation doesn't fail (as should be!)
  - the same requirement needed for **Vector**'s destructor (since **tmp** becomes destroyed at the end of the function before the return)

Destructors are critical for exception handling

- Destructors *should never fail*, i.e., never allow exceptions to escape from them

Exercise: Write a test program to show

what happens if an exception is thrown out from a destructor while the system is still propagating another?

- propagating an exception calls destructors (within the call stack)
- if such a destructor lets a new exception escape, the run-time system immediately terminates the program
- so a destructor should always catch potential exceptions (if any)
  - handle and recover from the exception -- or if not possible log out an error diagnostics and shut down the program
- the compiler cannot check, so it is the programmer's responsibility

Need for exception safety

Reusable library components vs. basic applications

- different levels of exception safety can be identified and are appropriate in different situations
- strong guarantee may be too expensive or not worth it:
  not all processing or programs can be made or need to be absolutely "failure safe"

For example

- an application program is not necessarily meant to be a separate reusable component (or a part of a library)
- when encountering an error, a simple application could report errors, decide to end its execution, discard all calculated results, and require the user to try it again with better and more valid input

Think again

- if you feel that you need a **delete** in your code: think again
  - prefer **delete**s in destructors
- if you feel that you need a **try-catch** block: think again
  - you might be able to do much better using RAII
- if you feel that you need a **new** operation..
  - prefer scope-controlled objects, or objects that are embedded "inline" within "owner objects"
  - use pointers only when need to replace a whole object "in place" (say, replace a buffer with a new longer one)
  - at least hide the dynamic allocation/deletion inside function members
    - a **vector** may reallocate its buffer but that is hidden

Summary

- write exception-safe class *libraries* and reusable *components*
- three different levels of exception safety can be provided: *basic*, *strong*, and *nofail*
- especially, the destructors require the *nofail* guarantee
- *exception neutrality* needed especially for templates (unknown type parameters with their unknown exceptions)
- play safe to prevent bugs and to debug
  - add redundant checks to verify assumptions
  - always initialize everything (especially pointers) to minimize random and unpredictable states
  - remember to clean up resources (usually via destructors)
  - instead of raw pointers use *smart pointers* (C++11)