Exception Handling in ANSI C++

Programming With Exceptions

Angelika Langer

Trainer/Consultant

`http://www.AngelikaLanger.com`

Programming With Exceptions

- Use of exceptions pervades an entire application and cannot be localized.
  - An exception can be propagated up the call stack.
  - Each exception "terminates" the respective current block.
- Throwing an exception is easy; writing code that uses a throwing function is hard.
  - We will see why.

Programming With Exceptions

- Exceptions can pop up everywhere.
- Before exception handling it was impossible to indicated errors in constructors, overloaded operators, and destructors.
  - Either they have no return code, or
  - the return code is used for purposes other than error reporting, e.g. operator chains

Exceptions Everywhere ...

A typical C idiom:

```
while (a[i++] = b[j++])
```

- **`a`** and **`b`** can be of different types, e.g. the STL containers **`vector`** and **`deque`.**
- **`i`** and **`j`** can be of different iterator types.
- Assignment can be overloaded for the element type.
- Converting constructors and cast operators can be involved.

Exceptions Everywhere ...

```
vector<string> a;              deque<char*> b;
vector<string>::iterator i;    deque<char*>::iterator j;
```

```
while (a[i++] = b[j++])
```

actually is a sequence of functions calls each of which might throw an exception:

```
while ((a.operator[](i.operator++()))
       .operator=(string
       (b.operator[](j.operator++())))))
```

Exceptions Everywhere ...

A typical C idiom:

```
while (a[i++] = b[j++])
```

If an exception appears ...

- where did it come from?

The order of evaluation of function arguments is unspecified. If an exception appears ...

- what are the current values of **`a`**, **`b`**, **`i`**, and **`j`**?

Programming With Exceptions

- Exceptions cannot be ignored.

- We must cope with them when they occur, even if we are not willing to handle them.
  - An exception terminates the current block,
  - current operations are aborted before they are finished,
  - objects might be left in inconsistent states, and
  - acquired local resources might not be released.

Exceptions cannot be ignored ...

```
class date {
public:   date(int d, int m, int y)
          :day(d), mon(m), year(y);

 friend istream&
 operator>>(istream& is, date& d)

 { return (is >> d.day >> d.mon >> d.year); }
};
```

An exception can leave the date object half-initialized.

- a typical problem when composite resources are manipulated

Exceptions cannot be ignored ...

```
template <class T>
void Stack<T>::push(const T& elem)
{ mutex_.acquire();
  v_[top_] = elem;
  top_++;
  mutex_.release();
}
```

In case of an exception the mutex object would not be released.

—a typical problem with dynamically acquired resources

Agenda

- **Resource Acquisition is Intialization**
- The `auto_ptr` template
- Exceptions in Constructors
- Exceptions in Destructors
- Preserve Exception Information
- Preserve the Object State
- An Exception-Safe `stack` Implementation
- Exception Safety

Resource Acquisition

```
void use_file (const char* filnam)
{ FILE* fil = fopen(filnam,"w");
  // use the file fil
  fclose(fil);
}
```

In case of an exception the file would not be closed.

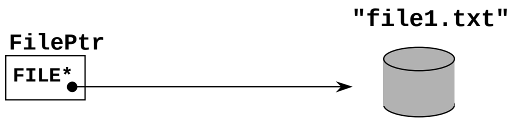
Resource Acquisition

```
void use_file (const char* filnam)
{ FILE* fil = fopen(filnam,"w");
  try {/* use the file fil */}
  catch (...)
  { fclose(fil);
    throw;
  }
  fclose(fil);
}
```

Resource Acquisition

- All excpetions are caught and the file is closed, i.e. the resource is released, in the **`catch`** block.
  - —Error-prone, because it can get rather complicated if numerous resources are acquired and released.
- A more elegant solution: Wrap resources into classes, and use constructors for acquisition and destructors for release.
  - —Destructors are called even when exceptions appear and this way release is guaranteed.

A File Pointer Class



```
class FilePtr {
private:
  FILE* fp_;
public:
  FilePtr (const char* filnam, const char* mod)
  : fp_(fopen(filnam,mod)) {  }
  FilePtr (FILE* fp) : fp_(fp) {  }
  ~FilePtr() { fclose(fp_); }
  operator FILE*() { fp_; }
};
```

Resource Acquisition

```
void use_file (const char* filnam)
{ FilePtr fil (filnam,"w");
    // use the file fil
}   // automatically closed via destructor
```

Agenda

- Resource Acquisition is Intialization
- **The `auto_ptr` template**
- Exceptions in Constructors
- Exceptions in Destructors
- Preserve Exception Information
- Preserve the Object State
- An Exception-Safe `stack` Implementation
- Exception Safety

Resource Acquisition

```
class Thing { /* ... */ };
void func ()
{ Thing* tp = new Thing;
  // ...
  delete tp;
}
```

In case of an exception the `Thing` would not be deleted.

The `auto_ptr` Class

- Use `auto_ptr` for dynamically allocated, local objects.
- An `auto_ptr` stores a pointer to an object obtained via new and deletes that object when it itself is destroyed (such as when leaving block scope).

Using `auto_ptr`

```
class Thing { /* ... */ };
void func ()
{ auto_ptr<Thing> tp(new Thing);
  // ...
}
```

`auto_ptr` takes care of deleting **`Thing`** when leaving the function body (either on normal return or when an exception appears).

The `auto_ptr` Class

```
template<class X> class auto_ptr {
private:
 X* ptr_;
public:   // construct/destroy:
 explicit auto_ptr(X* p =0) throw()
 : ptr_(p) {}

 ~auto_ptr() throw() { delete ptr_; }
};
```

The `auto_ptr` Class

The **`auto_ptr`** provides a semantics of strict ownership.

- An **`auto_ptr`** owns the object it holds a pointer to.
- Copying an **`auto_ptr`** copies the pointer and transfers ownership to the destination.
- If more than one **`auto_ptr`** owns the same object at the same time the behavior of the program is undefined.

Transfer of Ownership

```
auto_ptr<Thing> tp(new Thing);
auto_ptr<Thing> tp2 = tp;
```

- After assignment **`tp2`** owns the object, and **`tp`** no longer does.
- **`tp`** is empty; deleting **`tp`** would not delete any **`Thing`** object anymore.

Transfer of Ownership

```
Thing* p = new Thing;
auto_ptr<Thing> tp1(p);
auto_ptr<Thing> tp2(p);
```

Misuse:

- More than one **`auto_ptr`** owns the **`Thing`** object.

The `auto_ptr` Class

```
template<class X> class auto_ptr {
public:    // give up ownership:
 X* release() throw()
 { X* tmp = ptr_; ptr_ = 0; return tmp; }

public:    // copy constructor:
 auto_ptr(auto_ptr& a) throw()
 { ptr_(a.release()); }

};
```

The `auto_ptr` Class

More operations that give up ownership:

```
template<class X> class auto_ptr {
public:    // generic copy constructor:
 template<class Y>
 auto_ptr(auto_ptr<Y>&) throw();

public:    // generic conversion:
 template<class Y>
 operator auto_ptr<Y>() throw();
};
```

The `auto_ptr` Class

```
template<class X> class auto_ptr {
public:    //  change ownership:
 void reset(X* p=0) throw()
 { delete ptr_; ptr_ = p; }

public:    // assignment:
 auto_ptr& operator=(auto_ptr& a) throw()
 { if (&a!=this) reset(a.release()); }

 template<class Y> auto_ptr&
 operator=(auto_ptr<Y>&) throw();
};
```

The `auto_ptr` Class

```
template<class X> class auto_ptr {
public:   // members:
 X* get() const throw() { return ptr_; }

 X& operator*() const throw()
 { return *get(); }
 X* operator->() const throw()
 { return get(); }
};
```

The `auto_ptr` Class

The uses of **`auto_ptr`** include

- providing temporary exception-safety for dynamically allocated memory,
- passing ownership of dynamically allocated memory to a function, and
- returning dynamically allocated memory from a function.

`auto_ptr` cannot be used as the element type of the STL containers.

- **`auto_ptr`** does not meet the CopyConstructible and Assignable requirements for STL container elements.

Using auto_ptr

Conventional pointer member:

```
class X {
 T* pt_;
public:
 X() : pt_(new T) {}
 ~X(){ delete pt_; }
};
```

Alternative using **`auto_ptr`**:

```
class X {
 auto_ptr<T> apt_;
public:
 X() : apt_(new T) {}
 ~X() {}
};
```

Using auto_ptr

Container of pointers:

```
vector<T*> v1, v2;
v1 = v2;    // copies all pointers from  v2  to  v1
            // i.e. v1 and v2 share ownership of the pointed to
            // elements
```

Don't use **`auto_ptr`** with STL containers !!!

```
vector<auto_ptr<T> > v1, v2;
v1 = v2;    // copies all elements from  v2  to  v1,
            // i.e. v2 transfers ownership of all its elements to v1;
            // all auto_ptrs in v2  are emtpy after this assignment
```

Agenda

- Resource Acquisition is Intialization
- The auto_ptr template
- **Exceptions in Constructors**
- Exceptions in Destructors
- Preserve Exception Information
- Preserve the Object State
- An Exception-Safe `stack` Implementation
- Exception Safety

Exceptions in `new` Expressions

What happens if **`X`**'s constructor throws?

```
X* p1 = new X;
X* p2 = new X[256];
```

The memory allocated by the **`operator new()`** is freed. No memory leak!

Exceptions in Constructors

Constructors are a special case. If an exception propagates from an constructor ...

- the partial object that has been constructed so far is destroyed.
  - If the object was allocated with **`new`** the memory is deallocated.
- only the destructors of fully constructed subobjects are called.
  - The destructor of the object itself is not called.

Exceptions in Constructors

```
class X {
 S s_; T t_;
public:
 X(const S& s, const T& t)
 : s_(s), t_(t) // assume exception from copy ctor of T
 {}
 ~X(){}
};
```

Destructor for **`t_`** is *not* called, because it was not constructed.

Destructor for **`s_`** is called (fully constructed subobject).

Destructor **`~X()`** is *not* called.

Exceptions in Constructors

If a resource is obtained directly (not as part of a subobject) a resource leak can occur.

Only the allocation and construction of subobjects is reverted in case of an exception.

- No automatic cleanup for already performed initializations.

Exceptions in Constructors

```
class X {
 S* ps_; T* pt_;
public:
 X() : ps_(new S), pt_(new T) {}
 ~X(){ delete pt_; delete ps_; }
};
```

Assume an exception is thrown from the constructor of **`T.`**

Allocation of the temporary **`T`** object fails. Memory allocated with **`new T`** is deallocated; **`~T()`** is *not* called.

The pointers **`ps_`** and **`pt_`** are destroyed.

The construction of **`X`** fails; the destructor **`~X()`** is *not* called.

The object **`ps_`** points to is never deleted.

Exceptions from a Constructor Initializer List

How can we catch exceptions from a constructor initializer list?

```
X::X() try : ps_(new S), pt_(new T)
{}
catch(...)
{  // problem: don't know what happened
   // exception can stem from ctor initializer or function body
}
```

Exceptions in Constructors

A solution:

- Not ideal; error-prone in case of numerous dynamically acquired resources.

```
X::X(){
 try {ps_ = new S;}
 catch(...)
 { throw; /* do nothing, because no subobject is constructed yet */ }
 try {pt_ = new T;}
 catch(...)
 { delete ps_; }
}
```

Exceptions in Constructors

Another solution:

- Initialize pointers to 0, so that you can safely delete them.

```
X::X() : ps_(0), pt_(0)
{ try { ps_ = new S; pt_ = new T; }
   catch (...)
   { delete pt_;
     delete ps_; // reverse order
     throw;
   }
}
```

Exceptions in Constructors

Yet another solution: Use `auto_ptr`.

```
class X {
 auto_ptr<S> aps_;  auto_ptr<T> apt_;
public:
 X() : aps_(new S), apt_(new T) { }
 ~X() {}
};
```

Assume an exception is thrown from the constructor of `T.`

The subobject `apt_` is not created and need not be destroyed.

The subobject a`ps_` is destroyed; the destructor of a`ps_` destroys the object a`ps_` points to.

Rules

- Avoid resource leaks.
- Use "resource acquisition is initialization" for dynamically acquired resources.
  - Wrap resources into a class, acquire in its constructor, and release in its destructor.
- Use **`auto_ptr`** for dynamically allocated memory.

Agenda

- Resource Acquisition is Intialization
- The auto_ptr template
- Exceptions in Constructors
- **Exceptions in Destructors**
- Preserve Exception Information
- Preserve the Object State

Destructors and Exceptions

A destructor can be called

- as the result of normal exit from a scope, a `delete` expression, or an explicit destructor call, or
- during stack unwinding, when the exception handling mechanism exits a scope containing an object with a destructor.
  - —If an exception escapes from a destructor during stack unwinding `::std::terminate()` is called.

Destructors and Exceptions

- Do not let exceptions propagate out of a destructor!

```
X::~X()
try {  /* destructor body */  }
catch (...)
{ if (uncaught_exception())
        // This is an exception during stack unwinding.
        // Handle it! Do not re-throw!
  else
        // This is harmless. May propagate the exception.
}
```

Agenda

- Resource Acquisition is Intialization
- The auto_ptr template
- Exceptions in Constructors
- Exceptions in Destructors
- **Preserve Exception Information**
- Preserve the Object State
- An Exception-Safe `stack` Implementation
- Exception Safety

Rules

- Do not catch any exceptions if you do not know how to handle them.
  - Rewrite functions to preserve state instead of adding catch clauses.
  - If you cannot ignore propagated exceptions, use a catch-all clause.
  - If you get stuck, call **`terminate()`** instead of **`abort()`**.

Statement Rearrangement

Typical C++ code corrupts object state if assignment fails:

```
 array[i++] = element; // >>
```

Exception handling is expensive. Don't do this:

```
 try { array[i++] = element; } // >>
catch(...) { i--; throw; }
```

Rewrite to:

```
 array[i] = element; // >>
i++;
```

Rules

- Do not hide exception information from other parts of the program that might need them.
  - Always rethrow the exception caught in a catch-all clause.
  - Re-throw a different exception only to provide additional information.

Hiding Exceptions

```
template <class T> class Stack<T> {
public:
 struct AllocationError : public bad_alloc
 { size_t stack_size; } // has additional information
 Stack& operator=(const Stack& rhs)
 { // ...
   try { new_buffer = new T[new_elems]; }
   catch(...)
   { throw AllocationError(new_elems); }
   // ...
 }
};
```

What's wrong here?

Hiding Exceptions

```
try { new_buffer = new T[new_elems]; }
catch(...)
{ throw AllocationError(new_elems); }
```

What if **`T::T()`** throws an exception?

A caller's handler that is prepared to handle the constructor exception does not get a chance to do so, and a handler for the allocation error might try to solve the wrong problem.

Hiding Exceptions

A possible solution:

```
new_buffer = new(nothrow()) T[new_elems];
if (new_buffer == 0)
   throw AllocationError(new_elems);
```

Agenda

- Resource Acquisition is Intialization
- The auto_ptr template
- Exceptions in Constructors
- Exceptions in Destructors
- Preserve Exception Information
- Preserve the Object State
- **An exception-safe stack implementation**

A Stack Class

```
template<class T> class Stack {
 size_t nelems_;
 size_t top_;
 T* v_;
public:
 size_t count() const { return top_; }
 void push(T);
 T pop();
 Stack();
 ~Stack();
 Stack(const Stack&);
 Stack& operator=(const Stack&);
};
```

Exception-Safe `Stack::pop()`

- Identify all statements where an exception can appear.
- Identify all problems that can occur in presence of an exeption. On exit from the function:
  - Is the **`Stack`** object still unchanged?
  - Is it still in a valid, consistent state?
  - Is it still destructible?
  - Are there any resource leaks?
- Rewrite the function to meet the goals above!

The `Stack::pop()`

```
template <class T>
T Stack<T>::pop()
{
  if(top_==0)
    throw "pop on empty stack";
  return v_[--top_];
}
```

Possible Exception Sites

```
template <class T>
T Stack<T>::pop()
{
  if(top_==0)
    throw "pop on empty stack";
  // stack has not yet been modified
  // ok; nothing evil can happen here

  return v_[--top_];
}
```

Possible Exception Sites

```
template <class T> T Stack<T>::pop()
{ if(top_==0)    throw "pop on empty stack";
  return v_[--top_]; // >>
  // size_t decrement and array subscript- ok
  // return statement creates copy of element of type T
  // copy constructor of T - can fail
  // definitely a problem here!
}
```

Decrement happens before copy construction of return value.
The stack object is modified although the **`pop()`** operation fails.

Leave object un-modified

```
return v_[--top_]; // >>
// definitely a problem here!
//  The stack object is modified although the pop() operation fails.
```

```
try {  return v_[--top_];  }
catch(...)
{    // restore original state
     top_++;
     throw;
}
```

Rule

Leave your object in the state it had when the function was entered.

- Catch exceptions and restore the initial state.

Exception-Safe `Stack` Assignment

- Identify all statements where an exception can appear.
- Identify all problems that can occur in presence of an exeption. On exit from the function:
  - Is the **`Stack`** object still unchanged?
  - Is it still in a valid, consistent state?
  - Is it still destructible?
  - Are there any resource leaks?
- Rewrite the function to meet the goals above!

The Stack Assignment

```
template <class T>
Stack<T>& operator=(const Stack<T>& s)
{
  if(&s == this) return *this;
  delete[] v_;
  v_ = new T[nelems_ = s.nelems_];
  for (top_=0;top_<s.top_;top_++)
      v_[top_] = s.v_[top_];
  return *this;
}
```

Possible Exception Sites

```
template <class T>
Stack<T>& operator=(const Stack<T>& s)
{
  if(&s == this) return *this;
  // pointer comparison - ok
  // pointer copying for return - ok
  // ok; nothing evil can happen here


  // continued on next slide
}
```

Possible Exception Sites

```
template <class T>
Stack<T>& operator=(const Stack<T>& s)
{
  delete[] v_;
  // destruction of elements of type T, i.e. T::~T() is called
  // ok; if we assume that destructors do not throw
  // deallocation of heap memory - ok

  // continued on next slide
}
```

Possible Exception Sites

```
template <class T>
Stack<T>& operator=(const Stack<T>& s)
{
  v_ = new T[nelems_ = s.nelems_]; // >>
  // assignment of size_t objects - ok
  // allocation of heap memory - can fail!
  // construction of elements of type T - can fail!
  // pointer assignment - ok
  // definitely a problem here!

  // continued on next slide
}
```

Possible Exception Sites

```
template <class T>
Stack<T>& operator=(const Stack<T>& s)
{ for (top_=0;top_<s.top_;top_++)
   // assignment, comparison, increment of size_t objects - ok

      v_[top_] = s.v_[top_]; // >>
      // array subscript - ok
      // assignment operator for type T  - can fail!
      // definitely a problem here!

  return *this;
}
```

Possible Exception Sites

```
delete[] v_;
v_ = new T[nelems_ = s.nelems_]; // >>
// definitely a problem here!
```

Old array is deleted.
Allocation of new array failed.
Pointer `v_` is left dangling.
The `Stack` destructor will try to delete `v_` => disaster!

The `Stack` object is not even destructible any more!

Possible Exception Sites

```
delete[] v_;
v_ = new T[nelems_ = s.nelems_]; // >>
for (top_=0;top_<s.top_;top_++)
     v_[top_] = s.v_[top_]; // >>
     // definitely a problem here!
```

Stack object is invalid because copy has been done only partially.

Since the old Stack data is already deleted, we cannot leave the Stack in its original state.

A solution: Define a NIL object, which represents a valid, but not usable value. (NULL pointer, zero-size string, emtpy stack)

Keep `Stack` destructible

```
delete[] v_;
v_ = new T[nelems_ = s.nelems_]; // >>
```

// Pointer **`v_`** is left dangling. The **`Stack`** destructor will try to delete **`v_`** => disaster!

```
T* tp = v_;
v_ = 0;
delete tp;
v_ = new T[nelems_ = s.nelems_]; // >>
```

// The **`Stack`** destructor can safely delete **`v_`** .

Leave `Stack` in a valid NIL state

```
v_ = new T[nelems_ = s.nelems_]; // >>
for (top_=0;top_<s.top_;top_++)
     v_[top_] = s.v_[top_]; // >>
```

// Stack object is invalid because copy has been done only partially.

```
v_ = new T[s.nelems_]; // >>
top_=0; nelems_=0;
for (size_t i=0;i<s.top_;i++)
     v_[i] = s.v_[i]; // >>
nelems_ = s.nelems_; top_ = s.top_;
```

// Stack object is NIL, i.e. empty, if copy fails.

Leave `Stack` untouched

```
v_ = new T[nelems_ = s.nelems_]; // >>
for (top_=0;top_<s.top_;top_++)
     v_[top_] = s.v_[top_]; // >>
```

// Stack object is invalid because copy has been done only partially.

```
new_buffer = new T[s.nelems_]; // >>
for (size_t i=0;i<s.top_;i++)
     new_buffer[i] = s.v_[i]; // >>
swap(v_,new_buffer); delete [] new_buffer;
nelems_ = s.nelems_; top_ = s.top_;
```

// Stack object is not modified until copy is successfully completed.

Rule

Perform critical operations through temporaries.

- Modify the object only after successful completion.

Leave valid NIL objects if you can't preserve the original state.

- Set object state to NIL before a critical operation and set to final value afterwards, i.e. only in case of success.

Keep your objects destructible.

- Do not leave dangling pointer in your objects.
- Delete pointers through temporaries.

Eliminate Resource Leak

```
new_buffer = new T[s.nelems_]; // >>
for (size_t i=0;i<s.top_;i++)
     new_buffer[i] = s.v_[i]; // >>
swap(v_,new_buffer);
delete [] new_buffer;
```

The memory allocated for **`new_buffer`** is not deallocated.

=> resource leak!

An `auto_array_ptr` Class

- Implement an auto pointer that holds a pointer to an array of elements.
- Solve the resource leak problem in the Stack assignment using the auto array pointer.

An `auto_array_ptr` Class

```
template <class X> class auto_array_ptr {
 X* p_;
public:
 explicit auto_array_ptr(X* p=0) throw()
 : p_(p) {}
 auto_array_ptr(auto_array_ptr<X>& ap) throw()
 : p_(ap.release()) {}
 ~auto_array_ptr()  { delete[] p_; }
 void operator=(auto_array_ptr<X>& rhs)
 { if(&rhs!=this) reset(rhs.release()); }
 // ...
};
```

An auto_array_ptr Class

```
template <class X> class auto_array_ptr {
public:
 // ...
 X& operator*() const throw() { return *p_; }
 X* operator->() const throw() { return p_; }
 X& operator[](size_t i) const throw()
 { return p_[i]; }
 X* get() const throw() { return p_; }
 // ...
};
```

An auto_array_ptr Class

```
template <class X> class auto_array_ptr {
public:
 X* release() throw()
 { X* tp=p_; p_=0; return tp; }
 void reset(X* p=0)
 { X* tp=p_;
   p_=p;
   if (tp!=p) delete[] tp;
 }
 X* swap(X* p) throw()
 { X* tp=p_; p_=p; return tp; }
};
```

Eliminate Resource Leak

```
new_buffer = new T[s.nelems_]; // >>
for (size_t i=0;i<s.top_;i++)
     new_buffer[i] = s.v_[i]; // >>
swap(v_,new_buffer); delete [] new_buffer;
```

// The memory allocated for **`new_buffer`** is not deallocated.

=> resource leak!

```
auto_array_ptr<T> new_buffer(new T[s.nelems_]);
for (size_t i=0;i<s.top_;i++)
    new_buffer[i] = s.v_[i];
v_ = new_buffer.swap(v_);
```

Rules

- Leave your object in the state it had when the function was entered.
- Perform critical operations through temporaries.
- Leave valid NIL objects if you can't preserve the original state.
- Keep your objects destructible.
- Use auto pointers and "resource acqusition is initialization" to avoid resource leaks.
- Avoid side effects in critical operations.

Agenda

- Resource Acquisition is Intialization
- The auto_ptr template
- Exceptions in Constructors
- Exceptions in Destructors
- Preserve Exception Information
- Preserve the Object State
- An Exception-Safe `stack` Implementation
- **Exception Safety**

Exception Safety

A user of a function is interested in the guarantees the function can give when exceptions are propagated.

Document not only the pre- and post conditions and the "normal" effect of a function, but also its exception safety guarantees.

Exception Safety Guarantees

Level 0: No guarantee.

Part of the data the function tried to modify might be lost or corrupted. Access to the data might cause a program crash.

Level 1: Destructibility.

Part of the data might be lost or in an incosistent state. It is not possible to safely to access to the data. However, it is guaranteed that the data can be destroyed.

Level 2: No resource leaks.

All objects that the function modifies have their destructors called, either when f() handles the exception or when those objects' destructors are called.

Exception Safety Guarantees

Level 3: Consistency.

All objects are left in a consistent state, not necessarily the state before f() was entered, and not necessarily the state after normal termination. All operations on the data have well-defined behavior. No crashes, no resource leaks, safe access.

Level 4: Full commit-or-rollback.

All objects are left in the state they had before execution of f(). All data values are restored to their previous values.

References

The C++ Programming Language, 3rd Edition
Bjarne Stroustrup
Addison Wesley Longman, 1997

More Effective C++
Scott Meyers
Addison Wesley Longman, 1996

References

C++ Report

Ten Rules for Handling Exception Handling Sucessfully
Harald M. Müller, January 1996

Coping with Exceptions
Jack W. Reeves, March 1996

Exceptions and Standards
Jack W. Reeves, May 1996

Ten Guidelines for Exception Specification
Jack W. Reeves, July 1996

Exceptions and Debugging
Jack W. Reeves,
November/December 1996

Making the World Safe for Exception
Matthew H. Austern, January 1998

Contact Info

Angelika Langer

Training & Consulting
Object-Oriented Software Development in C++ & Java

Munich, Germany

Email: **info@AngelikaLanger.com**
http:**//www.AngelikaLanger.com**

last update: 06.11.2005 ,11:35