Programming with C++ Exceptions

Angelika Langer

Trainer/Consultant

http://www.AngelikaLanger.com

Agenda

- Motivation
- Coping with Exceptions
- Exception Specifications
- Uncaught Exceptions
- Designing Exceptions

Exception Handling in ANSI C++

Part 1: Motivation

Why Exception Handling ?

- Before exception handling it was impossible to indicate 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 are a uniform way of indicating errors in C++.
  - » Even language constructs and standard library operations throw exceptions.

Traditional Error Handling

Compare exceptions handling to traditional error handling techniques:

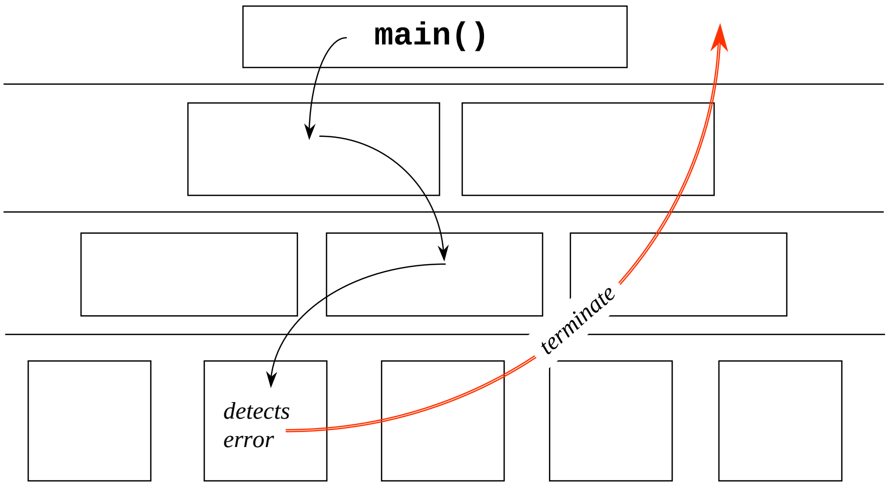
» terminate the program

» return an error code

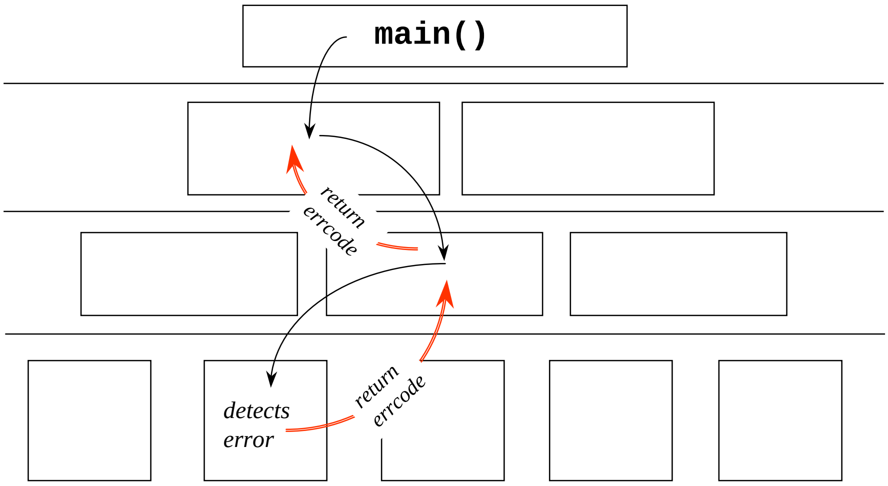
» return a legal value, but set a global error indicator (errno)

» call an error handler function

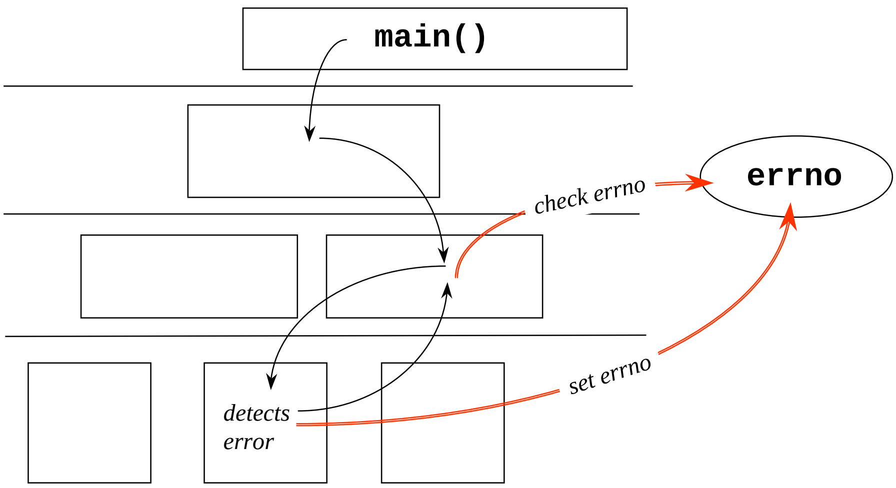
Termination



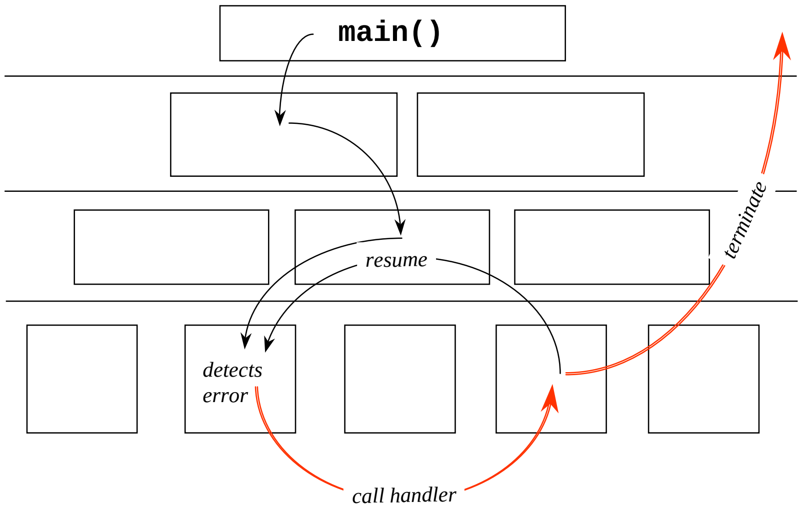
Return Error Codes



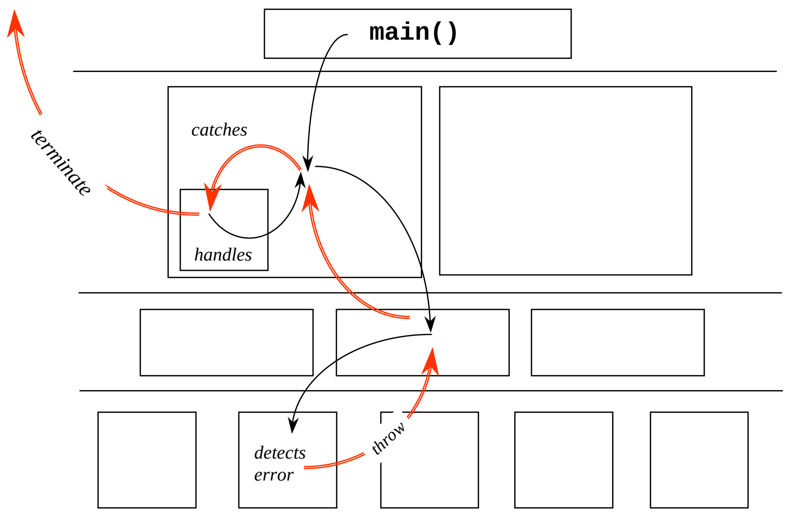
Global Error Indicator



Error Handler Function



Exception Handling



The Standard Exceptions

- In ANSI C++, the following language constructs throw exceptions:

C++ language	standard exception
`new`	**`bad_alloc`**
`dynamic_cast`	**`bad_cast`**
`typeid`	**`bad_typeid`**
exception specification	**`bad_exception`**

The Standard Exceptions

- In ANSI C++, several library components throw exceptions:

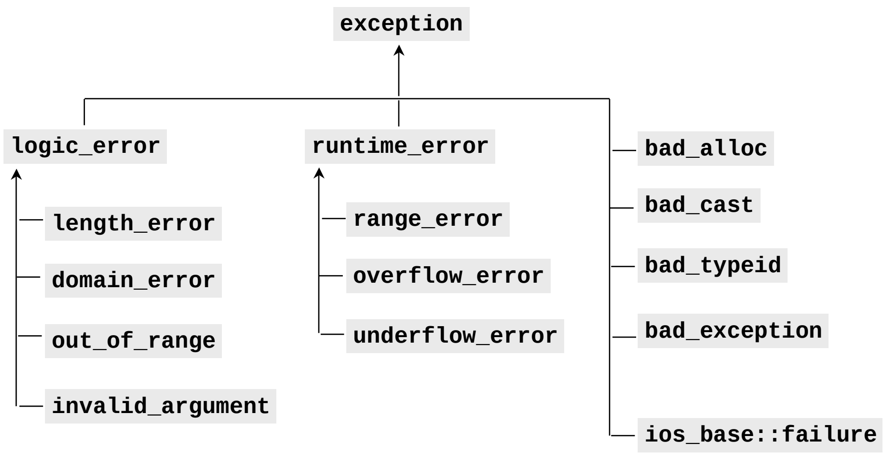
strings & sequential containers:
`out_of_range` & **`length_error`**

iostreams: **`ios_base::failure`**

locale: **`runtime_error`** & **`bad_cast`**

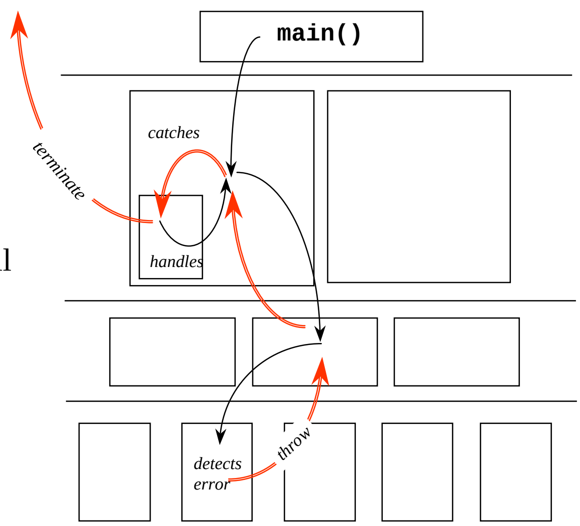
- Additionally, library components propagate any exceptions thrown by user-code they invoke.

The Standard Exception Hierarchy



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.



Programming With Exceptions

- Throwing an exception is easy; writing code that uses a throwing function is hard.

- 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

Exceptions Everywhere ...

```
vector<string> a;              deque<char*> b;
vector<string>::iterator i;    deque<char*>::iterator j;
i = a.begin();                 j = b.begin();
```

```
while (*i++ = *j++)
```

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

```
while ((i.operator*(i.operator++()))
       .operator=(string
       (j.operator*(j.operator++()))))
```

Exception Handling in ANSI C++

Part 2: Coping with Exceptions

Coping with Exceptions

- **Resource Acquisition is Intialization**
- The `auto_ptr` template
- Function `try` Blocks
- Exceptions in Constructors
- Exceptions in Destructors
- Some Guidelines
- Exception Safety Levels

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 exceptions 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
```


Coping with Exceptions

- Resource Acquisition is Intialization
- **The `auto_ptr` template**
- Function `try` Blocks
- Exceptions in Constructors
- Exceptions in Destructors
- Some Guidelines
- Exception Saftey Levels

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).

An **auto_ptr** manages an object on the heap.

Use of 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).

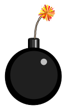
The auto_ptr Class

```
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_; }
};
```

Misuse

```
void foo() {
  static Thing t1;
  Thing t2;
  auto_ptr<Thing> tp1(&t1);
  auto_ptr<Thing> tp2(&t2);
}
```



Misuse:

- **auto_ptr** does not refer to a heap object.

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.

Compare to built-in pointers and smart pointers.

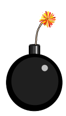
Transfer of Ownership

```
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.

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
```

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

```
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(); }
};
```

Coping with Exceptions

- Resource Acquisition is Initialization
- The `auto_ptr` template
- **Function `try` Blocks**
- Exceptions in Constructors
- Exceptions in Destructors
- Some Guidelines
- Exception Saftey Levels

Function try Blocks

function try block:

```
void f()
try { /* function body */ }
catch (...)
{ /* exception handler */ }
```

mostly equivalent to:

```
void f() {
   try { /* function body */ }
   catch (...)
   { /* exception handler */ }
}
```

Note:

- Flowing off the end of a function-try-block is equivalent to a **`return`** with no value; this results in undefined behavior in a value-returning function.
- The scope and lifetime of the parameters of a function extend into the handlers of a function-try-block.

Function try Blocks on Constructors

```
X::X(Arg a)
try : mem(0),Base(a)
{ /* constructor body */ }
catch (...)
{ /* exception handler */ }
```

Catches exceptions from the constructor body and the constructor initializer list, i.e. also from member and base class initializers.

Note:

- As usual in a failed constructor, the fully constructed base classes and members are destroyed.
- This happens before entering the handler.
- In the handler you cannot access any base classes or non-static members of the object.
- Arguments to the constructor are still accessible.

Function try Blocks on Constructors

- You cannot "handle" the exception and finish building the object.
- You can **NOT** "return" from the handler.
  - » You can only leave the handler via a `throw` statement.
  - » When you flow off the end of the handler, the exception is automatically re-thrown.

Function try Blocks on Constructors

- Are useful for mapping the exception to meet an exception specification:

```
class X {
 Y y_;
public:
 class Error {}; // nested exception class
 X::X(const Y& y) throw(X::Error)
 try : y_(y)
 { /* constructor body */ }
 catch (...) // catches possible exception from Y::Y
 { throw X::Error(); }
}
```


Function try Blocks on Destructors

```
X::~X()
try { /* destructor body */ }
catch (...)
{ /* exception handler */ }
```

Catches exceptions from the destructor body and from destructors of members and base classes.

- You cannot "handle" the exception and stop destruction of the object.
- You can "return" from the handler, but when control reaches the end of the handler, the exception is automatically re-thrown.

Function try Block on main()

```
int main()
try { /* body */ }
catch (...)
{ /* exception handler */ }
```

- Does not catch exceptions thrown by constructors or destructors of global variables.

Coping with Exceptions

- Resource Acquisition is Intialization
- The `auto_ptr` template
- Function `try` Blocks
- **Exceptions in Constructors**
- Exceptions in Destructors
- Some Guidelines
- Exception Saftey Levels

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.

Coping with Exceptions

- Resource Acquisition is Intialization
- The `auto_ptr` template
- Function `try` Blocks
- Exceptions in Constructors
- **Exceptions in Destructors**
- Some Guidelines
- Exception Saftey Levels

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.
}
```


Coping with Exceptions

- Resource Acquisition is Intialization
- The `auto_ptr` template
- Function `try` Blocks
- Exceptions in Constructors
- Exceptions in Destructors
- **Some Guidelines**
- Exception Safety Levels

Rules

Ideally, leave your object in the state it had when the function was entered.

» Catch exceptions and restore the initial state.

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&);
};
```

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.

Preserve the object state

```
template <class T> T Stack<T>::pop()
{ if (top_==0)
     throw "pop on empty stack";

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


Rules

- Do not catch any exceptions if you do not know how to handle them.
- Avoid **`catch`** clauses.
  - » Rewrite functions to preserve state instead of adding catch clauses.
- If you cannot ignore propagated exceptions, use a catch-all clause.

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

Keep your objects destructible.

» Do not leave dangling pointer in your objects.

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 and  pointer copying for return - ok

  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
  ...
}
```

Possible Exception Sites

```
template <class T>
Stack<T>& operator=(const Stack<T>& s)
{...
 delete[] v_;
 v_ = new T[nelems_ = s.nelems_]; // >>
  // allocation and construction - can fail!
  ...
}
```

- Old array deleted; allocation of new array fails.
- Pointer **v_** is left dangling.
- The **Stack** destructor will try to delete **v_** => disaster!
- The **Stack** object is not even destructible any more!

Keep Stack destructible

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

// Pointer **v_** is left dangling. The **Stack** object is not even destructible any more!

Rewrite to:

```
delete[] v_;
v_ = 0;  // The Stack destructor can safely delete v_ .
v_ = new T[nelems_ = s.nelems_]; // >>
```

Rules

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.

Perform critical operations through temporaries.

» Modify the object only after successful completion.

Possible Exception Sites

```
template <class T>
Stack<T>& operator=(const Stack<T>& s)
{…
 delete[] v_; v_ = 0;
 v_ = new T[nelems_ = s.nelems_]; // >>
 for (top_=0;top_<s.top_;top_++)
     v_[top_] = s.v_[top_]; // >>
     // assignment operator for type T - can fail!
}…
```

- 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.

Leave Stack in a valid NIL state

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

```
delete[] v_; v_ = 0;
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

Another solution: Use temporaries and modify the original only after successful completion.

```
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_;
```

Rules

- Avoid resource leaks.
  - Use auto pointers.
  - Implement an auto *array* pointer that holds a pointer to an array of elements.

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;
nelems_ = s.nelems_; top_ = s.top_;
```

What's wrong now?

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

=> resource leak!

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()); }
 // ...
};
```


Use auto array pointer

```
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_);
nelems_ = s.nelems_; top_ = s.top_;
```

Striving for Exception-Safety

- 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 object still unchanged?
  - Is it still in a valid, consistent state?
  - Is it still destructible?
  - Are there any resource leaks?

Coping with Exceptions

- Resource Acquisition is Intialization
- The `auto_ptr` template
- Function `try` Blocks
- Exceptions in Constructors
- Exceptions in Destructors
- Some Guidelines
- **Exception Safety Levels**

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 inconsistent 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 executin of f(). All data values are restored to their previous values.

Exception Handling in ANSI C++

Part 3: Exception Specifications

Exception Specifications

- **Language Feature and Intended Use**
- Problematic Uses
- Coping with Unexpected Exceptions

Exceptions Specifications

- Specify the set of exceptions that might be thrown by a function as part of the function declaration:

```
void f(int a) throw(bad_alloc,Matherr);
```

- **`f()`** throws only exceptions of type **`bad_alloc`**, **`Matherr`**, or of derived types.
- If the **`f()`** violates this guarantee the function **`::std::unexpected()`** is called, which by default calls **`terminate()`**.

Exception Specifications

```
void f() throw(X,Y)
{ /* ... */ }
```

is equivalent to:

```
void f()
try { /* ... */ }
catch (X) { throw; }
catch (Y) { throw; }
catch (...) { ::std::unexpected(); }
```


Exception Specifications ...

... allow *better documentation* of a function's behavior.

- The function declaration is accessible to a user,
- the function definition often is not.

```
int f();          // can throw any exception
int f() throw();  // does not throw exceptions
```

Exception Specifications

- Language Feature and Intended Use
- **Problematic Uses**
- Coping with Unexpected Exceptions

Exception Specifications ...

... cause problems.

- A no-throw specification keeps the compiler from propagating an exception.
- You still cannot ignore the exception when it occurs.
- You do not even get a chance to handle it either.

No-Throw Problem

```
void f() throw()
{ g(); }
```

Assume: **`g()`** allocates memory via **`operator new`**, which may throw **`bad_alloc`**, which leads to **`unexpected()`**, which aborts the program.

A caller of **`f()`** might be willing and prepared to handle **`bad_alloc`**, but the *exception* is *not* even *propagated*.

```
void h()
{ try{ f(); }
  catch (bad_alloc&)  // we never get here because f() has
                      // a throw() specification
  {/*...*/}
}
```


No Compile-Time Checking

The compiler should, but does not, check the following violation:

```
X get(int i) throw(out_of_range);

int find(X& x) throw()
{ // ...
  x == get(i);  // no compile-time error,
                // but run-time check, i.e. invocation
                // of unexpected()
}
```

No Compile-Time Checking

Reason:

- calls to legacy code that does not have exception specifications

What could a compiler possibly be doing?

- force legacy code to also have exception specifications
- force the programmer to catch all potential exceptions
- force the compiler to check the code of the called function
  - severe impact on compile-time and dependencies among components

No Compile-Time Checking

Here the compiler shall not and does not issue an error or warning:

```
X get(int i) throw(out_of_range);
int find(X& x) throw()
{ for (int i=0;i<size();++i)  // i cannot be out of range
  { if (x == get(i))  // no out of range exception can ever be raised here
        return i;
  }
   return -1;
}
```

Rules

- Use exception specifications to document what a function does, not what it is intended to do!
  - » Add them after implementation.
- Exception specifications can impair robustness of code.
  - » Avoid exception specifications of possible!
- Reserve a **`throw()`** specification for only the most trivial functions!

... impair robustness ...

Example:

- Function **`parse()`** is carefully designed not to raise any exception.
  - » For this reason it has a **`throw()`** specification.
- If its implementation is changed from use of **`char*`** to use of **`::std::string`** ...
  - » **`string`** can throw **`bad_alloc`**, which would call **`unexpected()`** => death penalty!
  - » BTW, the compiler does not warn about the violated guarantee!
  - » Callers with a **`bad_alloc`** handler do not even get a chance of handling the exception.

no-throw specification

Think of relinking with a new version of the C++ library where **`operator new`** suddenly throws **`bad_alloc`** ...

» In pre-exception handling C++, **`operator new`** returned 0 to indicate allocation failure.

» Standard C++'s **`operator new`** throws **`bad_alloc`**.

» Use the *nothrow new* if you do not want to add catch clauses to old code.

```
X* p1 = new X;              // throws bad_alloc if no memory
X* p2 = new(nothrow) X;// returns 0 if no memory
```


Templates and Exception Specifications

What would be a sensible exception specification for the following function template?

```
template <class Iterator, class Compare>
Iterator max_element
(Iterator first, Iterator last,Compare comp)
{ if (first == last) return first;
    Iterator result = first;
    while (++first != last)
          if (comp(*result, *first)) result = first;
    return result;
}
```

ρ Do not put exception specifications on template functions!

Virtual Functions

Redefined virtual functions must have an exception specification at least as restrictive as the base class version.

Otherwise a caller (seeing only the base class version) couldn't be expected to catch additional exceptions.

Virtual Functions

```
class B {
public:
  virtual void f(); // can throw anything
  virtual void g() throw(X,Y);
  virtual void h() throw(X);
};
class D : public B {
public:
  void f() throw(X);   // ok
  void g() throw(X);   // ok: D::g() is more restrictive
  void h() throw(X,Y); // error: D::h() is less restrictive
};
```

Problem with Virtual Functions

```
class Base {
 virtual void foo() throw(logic_error);
};
class derived : public Base {
 void foo() throw(logic_error, bad_alloc);
}
```

- No overriding function can use **`new`** unless it is willing to handle the exception.
- This is a problem to programmers deriving from Base.

Rules

- Put only general exception specifications on virtual functions!

```
class Base {
 virtual void foo() throw(LibraryException);
};
class derived : public Base {
 virtual void foo() throw(LibraryBadAlloc);
}
```

Organization of Exceptions and Specifications

- A well-designed subsystem shall have all its exceptions derived from a common base class.
- E.g. all standard library exceptions are derived from class **`exception`**.
- A function declared as

```
void f() throw(exception);
```

  will pass any **`exception`** to its caller.
- Thus, no **`exception`** in **`f()`** will trigger **`unexpected()`**.

Exception Specifications

- Language Feature and Intended Use
- Problematic Uses
- **Coping with Unexpected Exceptions**

Rules

If a function has an exception specification and the program cannot afford to abort,

install an unexpected-handler that maps the "bad" exception to a "good" one.

Intercepting unexpected()

- Consider a function f() written for a non-networked environment:

```
void f() throw(XYZerr);
```

- **`f()`** throws only exceptions of subsystem XYZ.
- Assume **`f()`** shall be called in a networked environment.
- It triggers **`unexpected()`** when a networking exception occurs.

  - Redefine **`f()`**, or
  - redefine **`unexpected()`**.

Installing an Unexpected-Handler

```
class Handler {
 PFV old;
public:
 Handler(PFV f)
 { old = ::set_unexpected(f); }
 ~ Handler()
 { ::set_unexpected(old); }
};
```


Redefine unexpected()

Mapping an unexpected exception into an expected one:

```
class XYZunexpected : XYZerr {};
void mapToXYZerr() throw(XYZunexpected)
{ throw XYZunexpected(); }

void networked_f() throw(XYZerr)
{ Handler h(&mapToXYZerr);
  f(); // a network exception triggers unexpected(),
       // which now throws an XYZunexpected,
       // which is derived from XYZerr and for this reason
       // "expected" in terms of the wrapper's exception specification
}
```

Recovering the Original Exception

The information about the original network exception is lost.

```
class XYZunexpected : XYZerr {
public: NetExc* e;
        XYZunexpected(NetExc* n) : e(n) {}
};
void mapToXYZerr() throw(XYZunexpected)
{ try { throw; }     // re-throw; to be caught immediately!
  catch(NetExc& e) { throw XYZunexpected(&e); }
  catch(...)       { throw XYZunexpected(0); }
}
```


Exception Handling in ANSI C++

Part 4: Designing Exceptions

Designing Exceptions

- **Error Handling Strategies**
- Design of Exception Classes

Error Handling Strategies

- Error indication and error handling are design issues.
  - Common strategies:
    - » return codes — *most common strategy*
    - » longjump / exit — *for fatal errors*
    - » errno — *often ignored*
    - » error callbacks — *rarely used; event based*
- How does exceptions handling fit in?

Handling of Local Errors

Return codes

- » must actively be checked
- » are mapped from one error code to another
- » have “local” scope, i.e. per level or component

Same strategy possible with exceptions:

- » catch exceptions
- » handle them or map them to other exception types

Handling of Fatal Errors

Fatal errors

- » traditionally “handled” via **`jongjmp`** or **`exit()`**
- » skip several levels in the call stack
- » have “global” scope, i.e. affect the entire application

Same strategy possible with exceptions:

- » let exceptions propagate up the call stack
- » counterpart to **`exit()`**: propagate out of **`main()`**
- » counterpart to **`longjmp`**: catch in a higher level component

Fatal Errors

Advantages of handling fatal errors via exceptions over **`longjmp`**/**`exit()`**:

» *automatic cleanup* thanks to destruction of local objects during stack unwinding

» fatal errors *cannot slip undetected*, uncaught exception terminates program

» *easy integration* of local and fatal errors

Localized Error Handling

Advantages of local error handling via exceptions over return codes:

» *no extra strategy* needed for functions without return codes (constructors, destructors, operators)

» *can't forget* to check return code; exception is propagated if ignored

» need not check the return of *every single call*, but can catch exceptions from a block of statements

» *fatal errors* need not be mapped from bottom to top, but are automatically propagated up to top

Localized Error Handling

Downsides of local error handling via exceptions:

- An exception not caught affects everybody up the call stack.

Exception specifications can help enforcing localized exception handling:

- An exception must be
  - caught & remapped or
  - caught & handled or
  - stays uncaught => program termination

Designing Exceptions

- Error Handling Strategies
- **Design of Exception Classes**

Design of Exceptions

Often: *provider*-centered exception design

» Component / library providers design exceptions according to their ideas.

Needed: *user*-centered exception design

» Requirements must come from those who must catch and cope.

- Which error information is needed?
- What shall the exception *type* express?
- Which information shall an exception *object* contain?
- What is the overall error handling strategy?

Exception Requirements

Determine which error information is required.

» severity level

- fatal or non-fatal?
- standard exceptions: runtime_error, logic_error

» exception safety level

- reuse or discard object? continue or terminate?

» origin / domain / component

- which component is in trouble?
- examples: NetError, IoError, DbError, ...

» problem description

- what's the problem? details, additional data, etc.
- examples: bad_alloc, bad_cast, illegal_hostid, file_not_found

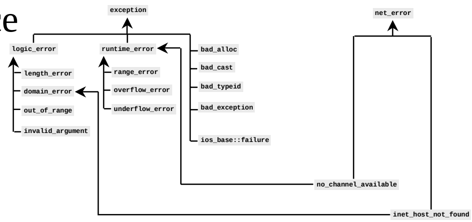
Providing Error Information

How do we express the error information -
via exception type or via exception state (or content) ?

	suggestions:
ρ severity level	*attribute, base type*
ρ exception safety level	*attribute, base type*
ρ origin / domain / component	*base type*
ρ problem description	*derived type, attribute*
ρ other	*attribute*

Design Considerations

- Derive from either **`logic_error`** or **`runtime_error`** to express a severity level.
- The domain base class should not be derived from class **`exception`**,
  - because **`exception`** is not a virtual base class of the standard exceptions.
- Consider multiple inheritance to express membership to a certain domain.



Designing Exception Classes

- Use the exception *type* to express information.
  - Remember: we throw objects, but we catch per type.
- Organize exceptions in a *hierarchy*.
  - easier to catch
    - can catch all derived exceptions per base class
    - need not know about all derived types
  - easier to extend
    - existing code not affected when derived exceptions are added
  - base type represents default
    - catch most derived type first
    - catch base type for default handling

Exception Hierarchy

A function that can handle network errors:

```
void g() { try { /* ... */ }
           catch(inet_host_not_found& e)
           { // handle error: try again with correct host id
             }
           catch(no_channel_available & e)
           { // fatal error: do local cleanup and let propagate exception
             throw; }
           catch(network_error& e)
           { // any other network error
              throw; }
}
```

Hierarchical Exception Design

`base exception` — general problem description; often used for specifying origin

↑

`derived exception` — specific problem description

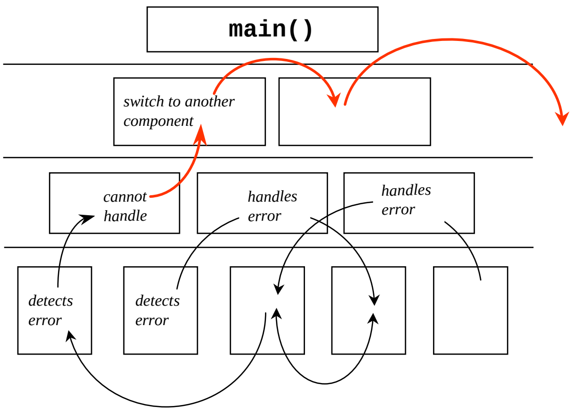
Base exception:

- » Does anybody want to catch the general error?
- » Is there a high level component that can react to the general exception?

Derived exception:

- » Specific exceptions are usually handled locally.

Catching Base Exceptions



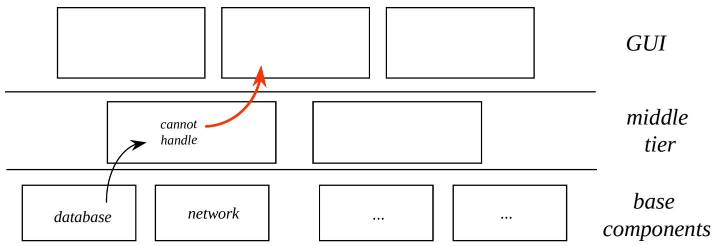
in case of fatal domain error, switch to another component

indicate and handle domain specific errors

Exceptions as Part of the Interface

- Exceptions are part of the interface of a component.
- Exchanging a components requires
  - » an completely new mapping of exception types, or
  - » affects everybody up the call stack

What if we switch from Oracle to Sybase ... ???



Using Exception Classes

- Give the user a way to avoid the exception.
  - » Supply a check function that can be used to make sure that an exception cannot occur.
  - » Should be supplied for all logic errors.
- Allow disabling of exceptions.
  - » global mask (e.g. exception mask in iostreams)
  - » additional argument (e.g. **`new(nothrow)`**)
  - » additional function (e.g. **`at()`** and **`operator[]()`**)

Exception Handling in ANSI C++

Part 6: References

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

The auto_ptr Class Template
Klaus Kreft & Angelika Langer,
November/December 1998

Contact Info

Angelika Langer

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

Munich, Germany

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