FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 12

Exceptions

Exceptions

- **exception handling**: Language-level support for managing run-time anomalies.
  - One independently developed section of code can detect and "raise" an exception that another independently developed part of the program can "handle."
  - The error-detecting part of the program throws an exception;
  - the error-handling part handles the exception in a catch clause of a try block.

C++ Standard library exceptions examples

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

exception	description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when fails with a referenced type
bad_exception	thrown when an exception type doesn't match any catch
bad_typeid	thrown by typeid
ios_base::failure	thrown by functions in the iostream library

Try and Catch

- **try block**: Block of statements enclosed by the keyword try and one or more catch clauses.
  - If the code inside the try block raises an exception and one of the catch clauses matches the type of the exception, then the exception is handled by that catch.
  - Otherwise, the exception is passed out of the try to a catch further up the call chain.
- **catch clause**: The part of the program that handles an exception.
  - A catch clause consists of the keyword catch followed by an exception specifier and a block of statements.
  - The code inside a catch does whatever is necessary to handle an exception of the type defined in its exception specifier.

Throwing

- **throw e**: Expression that interrupts the current execution path.
  - Each throw transfers control to the nearest enclosing catch clause that can handle the type of exception that is thrown.
  - The expression e is copied into the exception object.
- **Raise**: Often used as a synonym for throw.

Exception Object

- exception object: Object used to communicate between the throw and catch sides of an exception.
  - The object is created at the point of the throw and is a copy of the thrown expression.
  - The exception object exists until the last handler for the exception completes.
  - The type of the object is the type of the thrown expression.

Exception Specifier

- **exception specifier**: Specifies the types of exceptions that a given catch clause will handle.
  - An exception specifier acts like a parameter list, whose single parameter is initialized by the exception object.
  - Like parameter passing, if the exception specifier is a nonreference type, then the exception object is copied to the catch.
- **Terminate**: Library function that is called if an exception is not caught or if an exception occurs while a handler is in process. Usually calls abort to end the program.

Exception Handling I

- **Rethrow**: An empty throw—a throw that does not specify an expression.
  - A rethrow is valid only from inside a catch clause, or in a function called directly or indirectly from a catch.
  - Its effect is to rethrow the exception object that it received.
- **function try block**: A try block that is a function body.
  - The keyword try occurs before the opening curly of the function body and closes with catch clause(s) that appear after the close curly of the function body.
  - Function try blocks are used most often to wrap constructor definitions in order to catch exceptions thrown by constructor initializers.

Exception Handling II

- **catch-all**: A catch clause in which the exception specifier is (...).
  - A catch-all clause catches an exception of any type.
  - It is typically used to catch an exception that is detected locally in order to do local cleanup.
  - The exception is then rethrown to another part of the program to deal with the under-lying cause of the problem.

Exception Handling III

- **exception safe**: Term used to describe programs that behave correctly when exceptions are thrown.
- **stack unwinding**: Term used to describe the process whereby the functions leading to a thrown exception are exited in the search for a catch.
  - Local objects constructed before the exception are destroyed before entering the corresponding catch.

Exception Specification

- **exception specification**: Used on a function declaration to indicate what (if any) exception types a function throws.
  - Exception types are named in a parenthesized, comma-separated list following the keyword throw, which appears after a function's parameter list.
  - An empty list means that the function throws no exceptions.
  - A function that has no exception specification may throw any exception.
- **Unexpected**: Library function that is called if an exception is thrown that violates the exception specification of a function.

Exception Safety Guarantees

- **Basic guarantee**: Failed operations might alter programm state, but no leaks occur and affected objects/modules are still destructible and usable, in a consistent (but not necessarily predictable) state.
- **Strong guarantee**: Transactional commit/rollback semantics: Failed operations guarantee that program state is unchanged with respect to the objects operated upon and no side effects (iterators, ...).
- **Nofail guarantee**: Failure will not be allowed to happen, i.e. the operation will not throw an exception.

Constructor Failures

```
Class C { /*...*/ };

C c;  // <- Might emit an exception
```

The object’s lifetime never started and the object never came into existence.

```
C* pc = new C();  // <- Might emit an exception
```

Is there a memory leak?

Constructor Failures

```
Class C { /*...*/ };

C c;  // <- Might emit an exception
```

The object’s lifetime never started and the object never came into existance.

```
C* pc = new C();  // <- Might emit an exception
```

Is there a memory leak? No, the `new` operator takes care to deallocate the memory in case of an exception (the same is true for the array `new` operator!)

Exception Safe Code? (I)

```
void f( size_t N ) {
   float* array = new float[N];
   //... Use of array
   delete[] array;
}
```

Which exception safety guarantee does this code offer?

Exception Safe Code? (I)

```
void f( size_t N ) {
   float* array = new float[N];
   //... Use of array
   delete[] array;
}
```

Which exception safety guarantee does this code offer?

None! Anything can happen inbetween the allocation and deallocation of the array (return statements, exceptions, ...), leading to a memory leak!

Solution: Apply the RAII idiom!

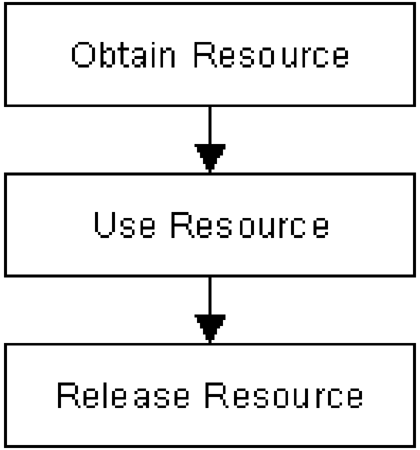
resource acquisition is initialization (RAII)

- In C++ RAII can be realized by smart pointers, e.g. the shared_ptr
-
```
class someResource{
    //internal representation holding pointers, handles etc.

public:
    someResource(){
        //Obtain resource.
    }

    ~someResource(){
    //Release resource.
    }

};
```



Exception Safe Code? (II)

```
// In some header file

void f( T1*, T2* );


// In some implementation file

f( new T1, new T2 );
```

Does this code have any potential exception safety problems?

Exception Safe Code? (II)

```
// In some header file

void f( T1*, T2* );


// In some implementation file

f( new T1, new T2 );
```

Does this code have any potential exception safety problems?

Yes! Compiler is free to order the function calls as necessary

1. Allocate memory for the T1
2. Construct the T1
3. Allocate memory for the T2
4. Construct the T2
5. Call f()

1. Allocate memory for the T1
2. Allocate memory for the T2
3. Construct the T1
4. Construct the T2
5. Call f()

Exception Safe Code? (II)

```
// In some header file

void f( T1*, T2* );


// In some implementation file

f( new T1, new T2 );
```

Does this code have any potential exception safety problems?

Yes! Compiler is free to order the function calls as necessary

1. Allocate memory for the T1	1. Allocate memory for the T1
2. Construct the T1	2. Allocate memory for the T2
3. Allocate memory for the T2	3. Construct the T1 ← Exception will lead to memory leak
4. Construct the T2	4. Construct the T2 ← Exception will lead to memory leak
5. Call f()	5. Call f()

Exception Safe Code? (III)

```
// In some header file

void f( shared_ptr<T1>, shared_ptr<T2> );


// In some implementation file

f(shared_ptr<T1>( new T1 ), shared_ptr<T2>( new T2 ) );
```

Does this code solve the exception safety problems?

Exception Safe Code? (III)


```
// In some header file

void f(shared_ptr<T1>, shared_ptr<T2> );


// In some implementation file

f(shared_ptr<T1>( new T1 ), shared_ptr<T2>( new T2 ) );
```

Does this code solve the exception safety problems?

No! The same problem still applies!

1. Allocate memory for the T1
2. Allocate memory for the T2
3. Construct the T1 ← Exception will lead to memory leak
4. Construct the T2 ← Exception will lead to memory leak
5. …

Exception Safe Code? (III)

A working solution to the problem:

```
// In some header file
void f(shared_ptr<T1>, shared_ptr<T2> );

// In some implementation file
shared_ptr<T1> t1( new T1 );
shared_ptr<T2> t2( new T2 );
f( t1, t2 );
```

Exception Safe Class Design (I)

Consider the following `Stack` class:

```
template< typename T >
class Stack {
 public:
   Stack();
   ~Stack();
   Stack( const Stack& );
   Stack& operator=( const Stack& );
   size_t count() const;
   void push( const T& );
   T pop();
 private:
   T* v_;
   size_t vsize_;
   size_t vused_;
};
```

Exception Safe Class Design (II)

Implementation of the default constructor: Is this exception safe?

```
template< typename T >
Stack<T>::Stack()
    : v_( new T[10] ) // <- possible memory leaks?
    , vsize_( 10 )
    , vused_( 0 )
{}
```

Exception Safe Class Design (II)

Implementation of the default constructor: Is this exception safe?

```
template< typename T >
Stack<T>::Stack()
    : v_( new T[10] ) // <- possible memory leaks?
    , vsize_( 10 )
    , vused_( 0 )
{}
```

This constructor is exception safe (and exception neutral, i.e. it propagates possible exceptions to the user)

Exception Safe Class Design (III)

Implementation of the destructor: Is this exception safe?

```
template< typename T >
Stack<T>::~Stack()
{
   delete[] v_;
}
```

Exception Safe Class Design (III)

Implementation of the destructor: Is this exception safe?

```
template< typename T >
Stack<T>::~Stack()
{
   delete[] v_;
}
```

The destructor is exception safe, given the T destructor cannot throw (but „destructors that throw are evil“)

Exception Safe Class Design (IV)

Helper function for copy construction and copy assignment:

```
template< typename T >
T* newCopy( const T* src,size_t srcsize, size_t destsize ) {
   assert( destsize >= srcsize );
   T* dest = new T[destsize];
   try {
      copy( src, src+srcsize, dest );
   }
   catch(...) {
      delete[] dest;    // This can't throw
      throw;            // Rethrow original exception
   }
   return dest;
}
```

Exception Safe Class Design (V)

Implementation of the copy constructor: Is this exception safe?

```
template< typename T >
Stack<T>::Stack( const Stack& other )
   : v_( newCopy( other.v_,
                  other.vused_,
                  other.vused_ ) )
   , vsize_( other.vused_ );
   , vused_( other.vused_ );
{}
```

Exception Safe Class Design (V)

Implementation of the copy constructor: Is this exception safe?

```
template< typename T >
Stack<T>::Stack( const Stack& other )
   : v_( newCopy( other.v_,
                  other.vused_,
                  other.vused_ ) )
   , vsize_( other.vused_ );
   , vused_( other.vused_ );
{}
```

With the help of the `newCopy` function the copy constructor is perfectly exception safe and neutral.

Exception Safe Class Design (VI)

Implementation of copy assignment: Is this exception safe?

```
template< typename T >
Stack<T>& Stack<T>::operator=( const Stack& other )
{
   if( this != &other ) {
      T* v_new = newCopy( other.v_, other.vused_, other.vused_ );
      delete[] v_;
      v_ = v_new;
      vsize_ = other.vused_;
      vused_ = other.vused_;
   }
}
```

Exception Safe Class Design (VI)

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Implementation of copy assignment: Is this exception safe?

```
template< typename T >
Stack<T>& Stack<T>::operator=( const Stack& other ) {
   if( this != &other ) {
      T* v_new = newCopy( other.v_, other.vused_, other.vused_ );
      delete[] v_;
      v_ = v_new;
      vsize_ = other.vused_;
      vused_ = other.vused_;
   }
}
```

With the help of the `newCopy` function the copy assignment operator is perfectly exception safe and neutral.

Exception Safe Class Design (VII)

Implementation of the `count` function: Is this exception safe?

```
template< typename T >
size_t Stack<T>::count() const
{
   return vused_;
}
```

There is absolutely no problem with this function.

Exception Safe Class Design (VIII)

Implementation of the `push` function: Is this exception safe?

```
template< typename T >
void Stack<T>::push( const T& t )
{
   if( vused_ == vsize_ ) {
      size_t vsize_new = vsize_*2+1;
      T* v_new = newCopy( v_, vsize_, vsize_new );
      delete[] v_;
      v_ = v_new;
      vsize_ = vsize_new;
   }
   v_[vused_] = t;
   ++vused_;
}
```

Exception Safe Class Design (VIII)

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Implementation of the `push` function: Is this exception safe?

```
template< typename T >
void Stack<T>::push( const T& t )
{
   if( vused_ == vsize_ ) {
      size_t vsize_new = vsize_*2+1;
      T* v_new = newCopy( v_, vsize_, vsize_new );
      delete[] v_;
      v_ = v_new;
      vsize_ = vsize_new;
   }
   v_[vused_] = t;
   ++vused_;
}
```

This function is exception safe and neutral.

Exception Safe Class Design (IX)

Implementation of the `pop` function: Is this exception safe?

```
template< typename T >
T Stack<T>::pop()
{
   if( vused_ == 0 ) {
      throw std::runtime_error( "pop from empty stack" );
   }
   else {
      T result = v_[vused_-1];
      --vused_;
      return result;
   }
}
```

Exception Safe Class Design (IX)

Implementation of the `pop` function: Is this exception safe?

```
template< typename T >
T Stack<T>::pop()
{
   if( vused_ == 0 ) {
      throw std::runtime_error( “pop from empty stack“ );
   }
   else {
      T result = v_[vused_-1];
      --vused_;
      return result;
   }
}
```

This is **NOT** exception safe: objects may get lost in case of exceptions!

Exception Safe Class Design (X)

The real problem: `pop` has two responsibilities:

- pop the top-most element
- return the just-popped value

Guideline: Prefer cohesion. Always endeavor to give each piece of code – each module, each class, each function – a single, well-defined responsibility (Single Responsibility Principle; SRP).

Exception Safe Class Design (XI)

Solution in the standard library: two functions (`top` and `pop`)

```
template< typename T >
T& Stack<T>::top() {
   if( vused_ == 0 )
      throw std::runtime_error( "empty stack" );
   return v_[vused_-1];
}

Template< typename T >
void Stack<T>::pop() {
   if( vused_ == 0 )
      throw std::runtime_error( "pop from empty stack" );
   else
      --vused_;
}
```

Exception Safe Class Design (XII)

Summary: Exception safety is never an afterthought. Exception safety affects a class‘s design. It is never “just an implementation detail“.

Exception Safe Functions

In order to be able to write exception safe code, at least two functions must provide the no-fail guarantee:

- the `swap` function
- destructors

Guideline: Provide a custom `swap` function if the default is not exception safe and never allow exceptions to escape your destructors!