

# Outline

- Handling exceptional cases: `errno`, `assert`, `longjmp`
- Goals of exception handling
- Handlers and exceptions
- Standard exceptions
- Exception safe programming
- C++11 `noexcept`
- `Exception_ptr`, `nested_exceptions`

# C Errno

```
#include <errno.h>

struct record { ... };
struct record rec;

extern int errno; /* preprocessor macro: thread-local since C11 */
int myerrno;      /* my custom error code */

FILE *fp;

if ( NULL == (fp = fopen( "fname", "r")) ) /* try to open the file */
{
    fprintf( stderr, "can't open file %s\n", "fname");
    fprintf( stderr, "reason: %s\n", strerror(errno)); /* perror(NULL) */
    myerrno = 1;
}
else if ( ! fseek( fp, n*sizeof(rec), SEEK_SET) ) /* pos to record */
{
    fprintf( stderr, "can't find record %d\n", n);
    myerrno = 2;
}
else if ( 1 != fread( &r, sizeof(r), 1, fp) ) /* try to read a record */
{
    fprintf( stderr, "can't read record\n");
    myerrno = 3;
}
else /* everything was successful up to now */
{
    ...
}
```

# C++ Errno

```
#include <cerrno>

struct record { ... };
struct record rec;

extern int errno; /* preprocessor macro: thread-local since C++11 */
int myerrno;      /* my custom error code */

FILE *fp;

if ( NULL == (fp = fopen( "fname", "r")) ) /* try to open the file */
{
    fprintf( stderr, "can't open file %s\n", "fname");
    fprintf( stderr, "reason: %s\n", strerror(errno)); /* perror(NULL) */
    myerrno = 1;
}
else if ( ! fseek( fp, n*sizeof(rec), SEEK_SET) ) /* pos to record */
{
    fprintf( stderr, "can't find record %d\n", n);
    myerrno = 2;
}
else if ( 1 != fread( &r, sizeof(r), 1, fp) ) /* try to read a record */
{
    fprintf( stderr, "can't read record\n");
    myerrno = 3;
}
else /* everything was successful up to now */
{
    ...
}
```

# iostream error handling

```
void f()
{
    std::ifstream file("input.txt");

    if ( ! file )    /* before C++11: void*, since C++11: bool */
    {
        std::cerr << "file opening failed\n";
        return -1;
    }

    for( int n; file >> n; ) /* while ( ! cin.fail() ) */
    {
        std::cout << n << '\n';
    }

    if ( file.bad() )
    {
        std::cerr << "i/o error while reading\n";
    }
    else if ( file.eof() )
    {
        std::cerr << "eof reached\n";
    }
    else if ( file.fail() )
    {
        std::cerr << "non-integer\n";
    }
}
```

# Assert

```
#include <cassert>    /* assert.h in C */

void open_file(std::string fname)
{
    assert(fname.length() > 0);

    std::ifstream f(fname.c_str());
    . . .
}
```

- Run-time error!

# Static assert (C++11)

```
#include <type_traits>

template <typename T>
void swap(T &x, T &y)
{
    static_assert( std::is_nothrow_move_constructible<T>::value, &&
                  std::is_nothrow_move_assignable<T>::value, "Swap may throw" );

    auto tmp = x;
    x = y;
    y = tmp;
}
```

# Goals of exception handling

- Type-safe transmission of arbitrary data from throw-point to handler
- Every exceptions should be caught by the appropriate handler
- No extra code/space/time penalty if not used
- Grouping of exceptions
- Work fine in multithreaded environment
- Cooperation with other languages (like C)

# Setjmp/longjmp

```
#include <setjmp.h>
#include <stdio.h>
```

```
jmp_buf x;
```

```
int main()
{
    int i = 0;

    if ( (i = setjmp(x)) == 0 )
    {
        f();
    }
    else
    switch( i )
    {
        case 1:
        case 2:
        default: fprintf( stdout, "error code = %d\n", i); break;
    }
    return 0;
}
```

```
#include <setjmp.h>
extern jmp_buf x;
```

```
void f()
{
    // ...
    g();
}

void g()
{
    // ...
    longjmp(x,5);
}
```



# Exceptions in C++

```
try
{
    f();
    // ...
}
catch (T1 e1) { /* handler for T1 */ }
catch (T2 e2) { /* handler for T2 */ }
catch (T3 e3) { /* handler for T3 */ }

void f()
{
    //...
    T e;
    throw e;    /* throws exception of type T */

    // or:
    throw T(); /* throws default value of T */
}
```

# Which handler?

**A handler of type H catches the exception of type E if**

- H and E is the same type
- H is unambiguous base type of E
- H and E are pointers or references and some of the above stands

# Exception hierarchies

```
class Base { ... };
class Der1 : public Base { ... };
class Der2 : public Base { ... };
class Der3 : public Der2 { ... };

try
{
    f();
    // ...
}
catch (Der3 &e1) { /* handler for Der3 */ }
catch (Der2 &e2) { /* handler for Der2 */ }
catch (Der1 &e3) { /* handler for Der1 */ }
catch (Base &e3) { /* handler for Base */ }

void f()
{
    if ( ... )
        throw Der3(); /* throw the most derived type */
}
```

# Exception hierarchies 2

```
class net_error { ... };
class file_error { ... };

class nfs_error : public net_error, public file_error { ... };

void f()
{
    try
    {
        ...
    }
    catch( nfs_error nfs ) { ... }
    catch( file_error fe ) { ... }
    catch( net_error ne ) { ... }
}
```

# Re-throw

```
class Base { ... };
class Der1 : public Base { ... };

void f()
try // function block can be try block
{
    g();
}
catch ( Base d ) // catch Base by value
{
    must_do_at_exception(b);
    throw; // re-throw original exception Der1
}
catch ( ... ) // catch all
{
    must_do_at_exception();
    throw; // re-throw original exception
}
```

# Exception hierarchies 3

```
#include <stdexcept>

struct matrixError
{
    matrixError( std::string r ) : reason(r) { }
    std::string reason;
    virtual ~matrixError() { }
};

struct indexError : public matrixError, public std::out_of_range
{
    indexError( int i, const char *r="Bad index" ) : matrixError(r), out_of_range(r), index(i)
    {
        std::ostringstream os;
        os << index;
        reason += ", index = ";
        reason += os.str();
    }

    const char *what() const noexcept override
    {
        return reason.c_str();
    }
    virtual ~indexError() { }
    int index;
};

struct rowIndexError : public indexError
{
    rowIndexError(int i) : indexError( i, "Bad row index" ) { }
};

struct colIndexError : public indexError
{
    colIndexError(int i) : indexError( i, "Bad col index" ) { }
};
```

- Catch by:
  - `std::logic_error`
  - `indexError`
  - Etc...

# std exception hierarchy

```
class exception {}; // <exception>
class bad_cast : public exception {}; // dynamic_cast
class bad_typeid : public exception {}; // typeid(0)
class bad_exception : public exception {}; // unexpected()
// class ios_base::failure : public exception {}; // before C++11
class bad_weak_ptr : public exception {}; // C++11 weak_ptr -> shared_ptr
class bad_function_call : public exception {}; // C++11 function::operator()
class bad_alloc : public exception {}; // new <new>
class bad_array_new_length : bad_alloc {} // C++11, new T[-1]

class runtime_error : public exception {}; // math. computation
class range_error : public runtime_error {}; // floating point ovf or unf
class overflow_error : public runtime_error {}; // int overflow INT_MAX+1
class underflow_error : public runtime_error {}; // int underflow INT_MIN-1
class system_error : public runtime_error {}; // e.g. std::thread constr.

class ios_base::failure : public system_error {}; // C++11 unexpected() <ios>

class logic_error : public exception {};
class domain_error : public logic_error {}; // domain error, std::sqrt(-1)
class invalid_argument : public logic_error {}; // bitset char != 0 or 1
class length_error : public logic_error {}; // length str.resize(-1)
class out_of_range : public logic_error {}; // bad index in cont. or string
class future_error : public logic_error {}; // C++11: duplicate get/set
```

# Exception specification before C++11

```
class E1;
class E2;

void f() throw(E1) // throws only E1 or subclasses
{
    ...
    throw E1();    // throws exception of type E1
    ...
    throw E2();    // calls unexpected() which calls terminate()
}
```

// same as:

```
void f()
try {
    ...
}
catch(E1) { throw; }
catch(...) { std::unexpected(); }
```



# Exception specification before C++11

```
class E1;
class E2;

void f() throw(E1, std::bad_exception) // throws only E1 or subclasses
{
    ...
    throw E1(); // throws exception of type E1
    ...
    throw E2(); // calls unexpected() which throws bad_exception()
}

typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler);
terminate_handler get_terminate(); // C++11

// until C++17
typedef void (*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler);
unexpected_handler get_unexpected(); // C++11

void f() throw() // not throws: can be optimized
```

# Noexcept operator in C++11

- **bool noexcept(expr);**
- Does not evaluate *expr* (similar to **sizeof** operator)
- False if
  - Expr throws
  - Expr has `dynamic_cast` or `typeid`
  - Has function which is `noexcept(true)` and not `constexpr`
- Otherwise **true**

# Noexcept specifier in C++11

## Replacing throw()

```
void f() noexcept(expr) { }  
void f() noexcept(true) { }  
void f() noexcept      { }
```

```
template <typename T>  
void f() noexcept ( noexcept( T::g() ) )  
{  
    T::g();  
}
```

# Destructors

## Destructors must not throw!

- Exception thrown during exception triggers **std::terminate()**
- Since C++11 every destructor is implicit **noexcept**
- It is possible to declare the destructor as **noexcept(false)**

## But the real situation is more complex:

<https://akrzemi1.wordpress.com/2011/09/21/destructors-that-throw/>

# Exception safety

```
class T1 { ... };  
class T2 { ... };  
  
template <typename T1, typename T2>  
void f( T1*, T2* );  
  
void g()  
{  
    f( new T1(), new T2() );  
    // ...  
}
```

## Scenario1

Allocates memory for T1  
Allocates memory for T2  
Constructs T1  
Constructs T2  
Calls f

## Scenario2

Allocates memory for T1  
Constructs T1  
Allocates memory for T2  
Constructs T2  
Calls f

# Exception safety

```
class T1 { ... };
class T2 { ... };

template <typename T1, typename T2>
void f( std::unique_ptr<T1>, std::unique_ptr<T1>);

void g()
{
    std::unique_ptr<T1> ptr1(new T1());
    std::unique_ptr<T2> ptr2(new T2());

    f( ptr1, ptr2 );
    // ...
}
```

# Exception safety in STL

- **Basic guarantee:** no memory leak or other resource issue
- **Strong guarantee:** the operation is atomic:  
it either succeeds or has no effect  
e.g. `push_back()` for vector, `insert()` for assoc. cont.
- **Nothrow guarantee:** the operation does not throw  
e.g. `pop_back()` for vector, `erase()` for assoc. cont., `swap()`

# Strong guarantee

```
template <class T>
class Vec
{
    // ...
private:
    size_t cap;
    size_t sz;
    T      *v;
};

Vec<T>& Vec<T>::operator=( const Vec& rhs)
{
    if ( this != &rhs )
    {
        delete[] v;
        cap = sz = rhs.sz;
        v = new T[cap];
        for (size_t i = 0; i < cap; ++i)
            v[i] = rhs.v[i];
    }
    return *this;
}
```



# Strong guarantee

```
template <class T>
class Vec
{
    // ...
private:
    size_t cap;
    size_t sz;
    T      *v;
};
```

```
Vec<T>& Vec<T>::operator=( const Vec& rhs)
```

```
{
    if ( this != &rhs )
    {
        delete[] v;
        cap = sz = rhs.sz;
        v = new T[cap];
        for (size_t i = 0; i < cap; ++i)
            v[i] = rhs.v[i];
    }
    return *this;
}

if ( this != &rhs )
{
    Vec tmp(rhs);
    cap = sz = tmp.sz;
    std::swap( v, tmp.v);
}
return *this;
}
```

# Exception safety in STL

	<b>vector</b>	<b>deque</b>	<b>list</b>	<b>map</b>
clear()	nothrow(copy)	nothrow(copy)	nothrow	nothrow
erase()	nothrow(copy)	nothrow(copy)	nothrow	nothrow
insert() one	strong(copy)	strong(copy)	strong	strong
insert() more	strong(copy)	strong(copy)	strong	strong
merge()			nothrow(comp)	
push_back()	strong	strong	strong	
push_front()		strong	strong	
pop_back()	nothrow	nothrow	nothrow	
pop_front()		nothrow	nothrow	
remove()			nothrow(comp)	
remove_if()			nothrow(pred)	
reverse()			nothrow	
splice()			nothrow	
swap()	nothrow	nothrow	nothrow	nothrow(cp,co)
unique()			nothrow(comp)	

# Other new features in C++11

- **class exception\_ptr** smart pointer type, default constructable, copyable, == if null or points to the same
- **make\_exception\_ptr(E e)** creates an exception\_ptr pointing to the exception object **e**.
- **current\_exception()** null ptr if called outside of exception handling or it returns an exception\_ptr pointing to the current exception
- **rethrow\_exception(std::exception\_ptr p)** rethrow exception **p**
- **class nested\_exception** polymorphic mixin class capture and store current exception  
has **rethrow\_nested() const** member function
- **throw\_with\_nested(T&& t)**  
**throw\_if\_nested(const E& e)**

# exception\_ptr

```
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>

void handle_eptr(std::exception_ptr eptr) // passing by value is ok
{
    try
    {
        if (eptr != std::exception_ptr())
        {
            std::rethrow_exception(eptr);
        }
    }
    catch(const std::exception& e)
    {
        std::cout << "Caught exception \"" << e.what() << "\"\n";
    }
}

int main()
{
    std::exception_ptr eptr;
    try
    {
        std::string().at(1); // this generates an std::out_of_range
    }
    catch(...)
    {
        eptr = std::current_exception(); // capture
    }
    handle_eptr(eptr);
} // destructor for std::out_of_range called here, when the eptr is destructed

// output: Caught exception "basic_string::at"
```

# Nesting exceptions

```
#include <iostream>
#include <stdexcept>
#include <exception>
#include <string>
#include <fstream>

void print_exception(const std::exception& e, int level = 0) // prints the string of an exception.
{
    I // if nested, recurses
    std::cerr << std::string(level, ' ') << "exception: " << e.what() << '\n';
    try {
        std::rethrow_if_nested(e);
    } catch(const std::exception& e) {
        print_exception(e, level+1);
    } catch(...) {}
}

void open_file(const std::string& s) // catches an exception and wraps it in a nested exception
{
    try {
        std::ifstream file(s);
        file.exceptions(std::ios_base::failbit);
    } catch(...) {
        std::throw_with_nested( std::runtime_error("Couldn't open " + s) );
    }
}

void run() // sample function that catches an exception and wraps it in a nested exception
{
    try {
        open_file("nonexistent.file");
    } catch(...) {
        std::throw_with_nested( std::runtime_error("run() failed") );
    }
}

int main() // runs the sample function above and prints the caught exception
try {
    run(); // exception: run() failed
} catch(const std::exception& e) { // exception: Couldn't open nonexistent file
    print_exception(e); // exception: basic_ios::clear
}

}
```