# 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

```c
#include <errno.h>
#include <stdio.h>   /* fopen */
#include <string.h>  /* strerror */

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>
#include <cstdio>   // std::fopen
#include <cstring>  // std::strerror

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

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

std::FILE *fp;

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

    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;
}
```

```
#if __STDC_HOSTED__ != 1
#  error "Not a hosted implementation"
#endif

#if __cplusplus >= 202302L
#  warning "Using #warning as a standard feature"
#endif
```

# 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

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

jmp_buf x;

int main()
{
    int i = 0;

    if ( (i = setjmp(x)) == 0 ) // try
    {
        f();
    }
    else    // catch
    {
        switch( i )
        {
        case  1: handler1(); break;
        case  2: handler2(); break;
        default: fprintf( stdout, "error code = %d\n", i); break;
        }
    }
    return 0;
}
```

```c
// perhaps in another source file

#include <setjmp.h>
extern jmp_buf x;

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

void g()
{
    if ( something_wrong() )
    {
        longjmp(x,2);  // throw
    }
}
```

# Setjmp/longjmp

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

jmp_buf x;

int main()
{
    int i = 0;

    if ( (i = setjmp(x)) == 0 ) // try
    {
        f();
    }
    else    // catch
    {
        switch( i )
        {
        case  1: handler1(); break;
        case  2: handler2(); break;
        default: fprintf( stdout, "error code = %d\n", i); break;
        }
    }
    return 0;
}
```

```c
// perhaps in another source file

#include <setjmp.h>
extern jmp_buf x;

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

void g()
{
    if ( something_wrong() )
    {
        longjmp(x,2);  // throw
    }
}
```

# Exceptions in C++

```
try  // dangerous area
{
  f();   // someth
}
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 */
}
```

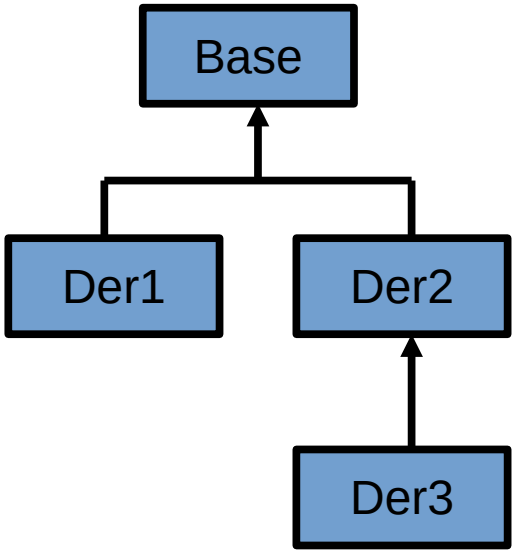
# Which handler?

**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

- Resolved run-time
- Not overloading!




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

try
{
  f();
  // …
}
catch (Der3 &e1)  { /* handler for Der3 */ } /* the most derived handler first */
catch (Der2 &e2)  { /* handler for Der2 */ }
catch (Der1 &e3)  { /* handler for Der1 */ }
catch (Base &e3)  { /* handler for Base */ } /* the base handler last */

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

}
```

# Exception hierarchies 2

- Multiple inheritance is possible
- And sometimes even useful

```
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

- Re-throw works only inside a catch block
- Re-throws the original object, not the (possible sliced) one

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

void g()
{
  throw Der1;  // throw derived exception Der1
}

void f()
try  // function block itself can be try block
{
  g();
}
catch ( Base b )  // catch Base by value: copied, since C++11 can be moved
{
  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 // perhaps member in class Matrix
{
    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 {};                                  // in <exception>

class bad_exception : public exception {};           // calls unexpected()
class bad_weak_ptr : public exception {};            // C++11 weak_ptr -> shared_ptr
class bad_function_call : public exception {}; // C++11 function::operator()
class bad_typeid : public exception {};              // typeid(0)
class bad_cast   : public exception {};              // dynamic_cast
  class bad_any_cast : public bad_cast {};           // C++17
class bad_variant_access : exception {}              // C++17
class bad_optional_access : exception {}             // C++17
class bad_alloc  : public exception {};              // new  <new>
  class bad_array_new_length : bad_alloc {}          // C++11, new T[-1]

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 container or string
  class future_error : public logic_error {}; // C++11: promise abandons the shared state

class runtime_error : public exception {};
  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
    class filesystem::filesystem_error : public system_error {}; // C++17
    class nonexistent_local_time : public system_error // C++20
    class ambigous_local_time : public system_error    // C++20
    class format_error : public system_error           // C++20
```

# 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(); }
```

- Unexpected before C++17  != unexpected since C++23
- Deprecated since C++11, removed since C++17

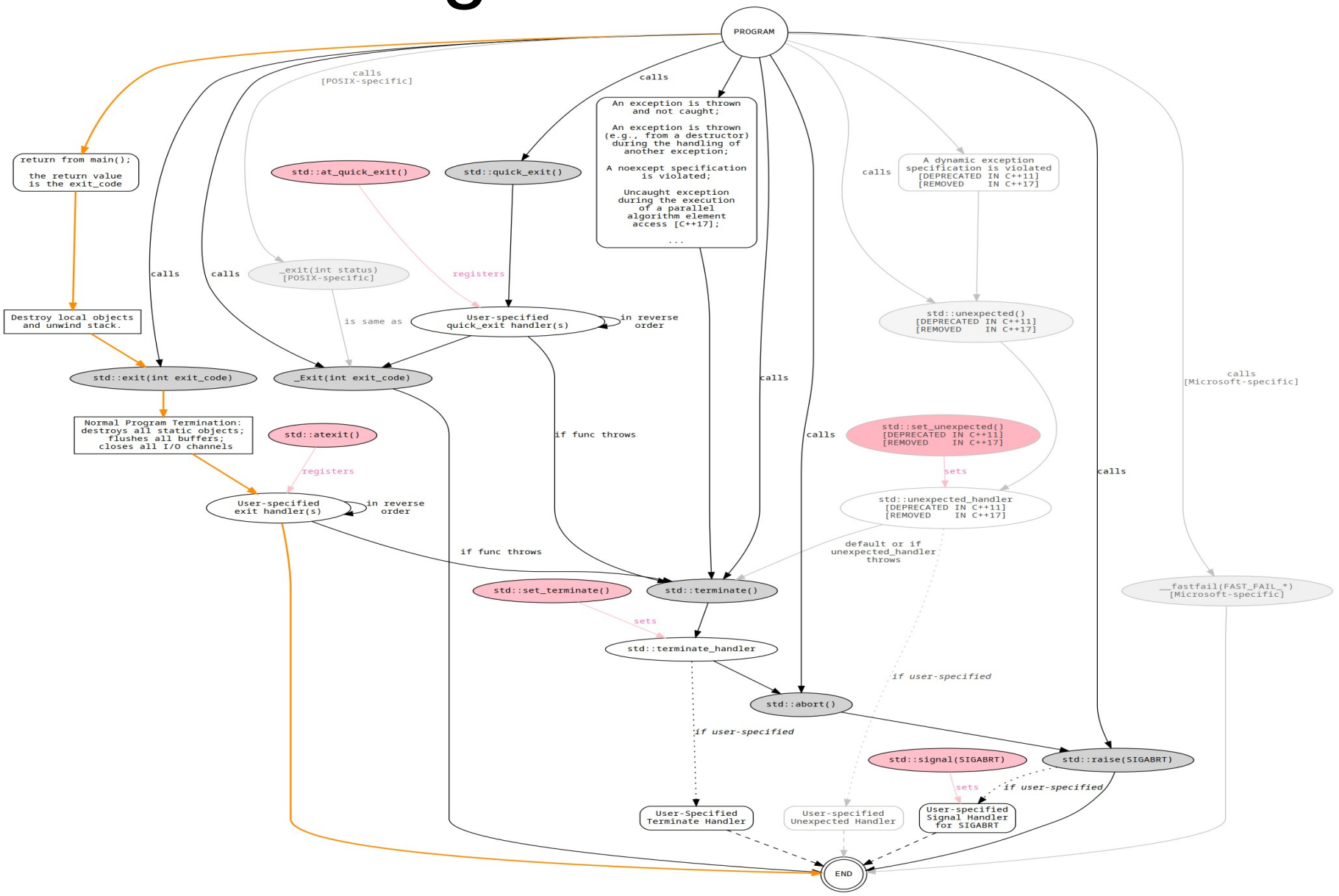
# Exception specification before C++11

```
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()   // does not throw, can be optimized`

- Deprecated since C++11, removed since C++17

# Program termination




https://github.com/adishavit/Terminators/blob/master/README.md

# Noexcept specifier in C++11

- Since C++11
  - Part of function type since C++17
  - But not part of of function signature (so no overload on noexcept)
- C++ functions are either non-throwing or potentially throwing
- C++ cannot execute full compile time check on possible exceptions
  - Partially due to possible call of non C++ functions
- Destructors, deallocation functions are non-throwing
- Implicitly declared or defaulted default-, copy- and move constructors, copy, move oper.
  - Unless base class or called operations throw
- Noexcept is important for optimizations and program safety

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

# Noexcept operator in C++11

- **bool noexcept(expr);**
- Can be used inside function template noexcept specifier
- Compile-time check: does not evaluate *expr* (like **sizeof**)
- False if
  - Expr throws
  - Expr has dynamic_cast or typeid
  - Has function which is no noexcept(true) and not constexpr
- Otherwise **true**

```
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)**
- If exception is thrown during stack unwinding: **std::terminate** is called instead
- 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 (before C++17)**

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

- Usually solved by rearranging the source and use sequence points
- Since C++17, parameter evaluation order changed

undefined -> unspecified

```
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);
  }
  // tmp deleted here with the old buffer

  return *this;
}
```

# Exception safety in STL

| | vector | deque | list | map |
|---|---|---|---|---|
| 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) | |

# Some new features in C++11

- **class exception_ptr** smart pointer type,default constructible, 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 {                                                   // exception: run() failed
      run();                                            //  exception: Couldn't open nonexistent file
    } catch(const std::exception& e) {                  //   exception: basic_ios::clear
        print_exception(e);

}
```

# Optional (C++17)

- Maybe monad implementation
- Replaces return types like **std::pair<T,bool>**
- Optional contains value
  - Initialized/assigned with value of T
  - Initialized/assigned with **optional<T>** which contains value
- Optional does not contain value
  - Default initialized or initialized with value of **std::nullopt_t**
  - Initialized/assigned with **optional<T>** which does not contain value
- If **optional<T>** contains a value, than it is allocated as T
  - Not a pointer based heap storage
- Convertible to **bool**: true if contains value
- No optional reference

# std::optional

```
std::optional<int> convert( const std::string& s)
try
{
    return std::stoi(s);  // C++11
}
catch (std::invalid_argument e)  // s is not an integer
{
    return {};   // std::optional<int>{std::nullopt}
}
catch (std::out_of_range e)   // result cannot be represented in int
{
    return {};   // std::optional<int>{std::nullopt};
}

int main()
{
    int i = convert("42").value_or(-1);
}
```

# Use of optional

```
void f(bool b1)
{
  std::optional<int> opt1;  // default constr: std::nullopt
  std::cout << opt1.value_or(-1) << '\n';   // -1
  try
  {
    std::cout << opt1.value() << '\n';  // throw std::bad_optional_access
  }
  catch( std::bad_optional_access& e)
  {
    std::cerr << e.what() << '\n';
  }
  opt1 = b1 ? std::optional<int>(42) : std::nullopt;  // 42

  std::cout << opt1.value_or(-1) << '\n';  // 42
  if ( opt1 )  // true
  {
    std::cout << opt1.value() << '\n';  // 42
    *opt1 = 2;   // access contained data, also ->  exists
    int i = opt1.value();
    std::cout << i << '\n';   // 2
  }
}

-1
bad optional access
42
42
2
```

# Use of pointers

```
void f(bool b1)
{
  std::optional<std::string> opt2;  // std::nullopt
  *opt2 = "Hello";   // undefined behavior if std::nullopt

  std::cout << *opt2 << '\n';
  std::cout << std::boolalpha << opt2.has_value() << '\n';   // false

  std::cout << opt2.value_or("no value") << '\n';   // “no value”
  std::string s = *std::move(opt2);

  std::cout << s << ", " << opt2->size() << '\n';
}
```

```
Hello
false
no value
Hello, 0
```

# Expected (C++23)

Always holds either value_type or unexpected_type

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0323r12.html

https://youtu.be/PH4WBuE1BHI (Andrei Alexandrescu CppCon 2018)

```
#include <expected>  // since C++23

template <class T, class E>    // T can be void, but T must not be unexpected<>
class expected {
  // types
  using value_type = T;
  using error_type = E;
  using unexpected_type = std::unexpected<E>;
  template <class U> using rebind = expected<U, error_type>;

  // accessors
  bool has_value()
  operator bool()
  operator void()   // if T == void
  T*   operator->() // undefined behavior if not expected
  T&   operator*()  // undefined behavior if not expected
  void operator*()  // if T == void, undefined behavior if not expected
  T&    value()     // may throw std::bad_expected_value<E>
  E&    error()     // undefined behavior if expected
  T value_or(U def)
};
```