

Enhancing Exception Handling And Debugging Using C++23 `std::stacktrace`

EREZ STRAUSS

— Legal —

All Statements and representations are my own and do not reflect those of my employer.

The work presented here is my own and not that of my employer and I take sole responsibility for the work presented during this event.

// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NON INFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.

C++ Exception Handling - Which Call Sequence

```
#include <print>
#include <exception>
void bar() {
    throw std::runtime_error("Error 123");
}
void foo1() {
    std::println("foo1");
    bar();
}
void foo2() {
    std::println("foo2");
    bar();
}
int main(int argc, char**) {
    try {
        if (argc % 2) foo1();
        else foo2();
    }
    catch (const std::runtime_error& e) {
        std::println("Caught exception: {}",
            e.what());
    }
}
```

\$./test1

Caught exception: Error 123

\$

No info which foo() was called 🙄

C++ Exception Handling - Enhanced Exception

```
#include <print>
#include <exception>
void bar() {
    throw std::runtime_error("Error 123");
}
void foo1() {
    std::println("foo1");
    bar();
}
void foo2() {
    std::println("foo2");
    bar();
}
int main(int argc, char**) {
    try {
        if (argc % 2) foo1();
        else foo2();
    }
    catch (const std::runtime_error& e) {
        std::println("Caught exception: {}",
            e.what());
    }
}
```

```
$ LD_PRELOAD=./libcatch.so ./test1
Early init setting terminate handler
Starting catch block for exception:
0# __cxa_throw at
40_exception_handler/libcatch.cpp:78
1# bar() at 40_exception_handler/test1.cpp:5
2# foo1() at 40_exception_handler/test1.cpp:8
3# main at 40_exception_handler/test1.cpp:16
4# __libc_start_call_main at
../sysdeps/nptl/libc_start_call_main.h:58
5# __libc_start_main_impl at ../csu/libc-start.c:360
6# _start at :0
7#
```

throw_type: std::runtime_error

Caught exception: Error 123 😊

Agenda

- Current C++ features
 - C++ exception handling
 - C++20 `std::source_location`
 - C++23 `std::stacktrace`
- How to provide an enhanced exception handling?
- Windows example
- Looking ahead
 - C++26 exceptions with stacktrace
 - throwinfo - project next steps
 - other similar libraries

C++ Exception Handling

```
try { .... }  
  
throw exception;  
  
catch(type& e) { ... }
```

Example Code

```
using namespace std;

void funcC(const auto& param){
    string s = param + " C";
    println ("C: {} {} {}", s,
            (void*)&param, (void*)&s);
    throw s;
}

void funcB(const auto& param){
    string s = param + " B";
    println ("B: {} {} {}", s,
            (void*)&param, (void*)&s);
    funcC(s);
}

void funcA(const auto& param){
    string s = param + " A";
    println ("A: {} {} {}", s,
            (void*)&param, (void*)&s);
    funcB(s);
}

int main()
{
    try {
        const string s {"main"};
        funcA(s);
    }
    catch(string& se) {
        println("got string
exception: '{}'", se);
    }
    catch(...) {
        println("got unknown
exception");
    }
}
```

Memory layout High Address

```
Frame: main()
    Local string s
    funcA parameters
    saved registers

frame: funcA
    funcA - return address
    Local string s
    funcB parameters
    saved registers

frame: funcB
    funcB - return address
    Local string s
    funcC parameters
    saved registers

...
```

Exceptions - The Good Things

- Simplify the optimistic code
- Reduces conditions, makes the code more readable
- Centralized error handling
- There is a hierarchy of predefined exceptions `std::exception`, `std::runtime_error`, `std::logical_error`

The Limitations

- Multiple call sequences can cause same exception
- Catch blocks are called when the stack trace is already unwinded
- Logging from catch block is not enough for further debugging of the specific call
- debugging issue - setting a breakpoint at the catch block doesn't help
- Uncaught exception - process aborts

The unknown call sequence

```
void funcZ(int i) {  
    throw std::runtime_error(  
        "Error: " + std::to_string(i));  
}  
void funcB1(int i) {  
    funcZ(i + 10);  
}  
void funcB2(int i) {  
    funcZ(i - 10);  
}  
void funcA(int i) {  
    if ( i < 50 )  
        funcB1(i);  
    else  
        funcB2(i);  
}  
  
int main() {  
    try {  
        auto randomNumber  
            = make_random<int>();  
        std::println("Random number: {} {}",  
            randomNumber, randomNumber < 50);  
        funcA(randomNumber);  
    }  
    catch (std::exception& e) {  
        std::println("exception: {}",  
            e.what());  
    }  
}
```

Exceptions Wish List

Additional support throwinfo - Exception handling wish list

- Capture the full stacktrace with filenames, line number and function name
 - If thrown object type was not found in the available catch blocks parameters, handle it with some generic handler, before abort
 - If there are many thread - hold information whether their exceptions were on the fly or caught.
- ++ Exception Handling
- enable/disable these features per thread
 - Enable breakpoint at any throw location

C++20 `std::source_location`

- `source_location` compile time values.
- internally 8 bytes reference to const struct
- four accessors `file_name()`, `line()`, `function_name()`, `column()`
- `source_location::current()` - compile time value
- no dependency on compiler flags

C++20 std::source_location

```
#include <print>
#include <source_location>
#include <string>

template<> struct std::formatter<std::source_location> : std::formatter<std::string> {
    template<typename FormatContext>
    auto format(const std::source_location& loc, FormatContext& ctx) const {
        std::string s = std::format("{}: {}: {} {}",
            loc.file_name(), loc.line(), loc.column(),
            loc.function_name());
        return std::formatter<std::string>::format(s, ctx);
    }
};

void report(const auto&& v, std::source_location sloc = std::source_location::current()) {
    std::println("report: {} from: {}", v, sloc);
}
```

C++20 std::source_location

```
void foo() {
    std::source_location sloc{};
    std::println("foo: Source location: {}", sloc);
}

void bar() {
    std::source_location sloc{std::source_location::current()};
    std::println("bar: Source location: {}", sloc);
}

int main() {
    using namespace std::literals;
    foo();
    bar();
    report(4);
    report(5.6);
    report("this is a text message, right"s);
}

$ ./example source_location
foo: Source location: :0:0
bar: Source location: example_source_location.cpp:30:31 void bar()
report: 4 from: example_source_location.cpp:39:5 int main()
report: 5.6 from: example_source_location.cpp:40:5 int main()
report: this is a text message, right from: example_source_location.cpp:41:5 int main()
```

C++ Runtime Stack

- Multiple frames - one for each function in the current call sequence
- Holds local variables of current function - initialized if using RAI
- Holds return addresses
- stack walking - finding the return addresses for each function call
- stack unwinding - call the destructors of the objects on the stack
- In debugger - use up / down to see different frames

C++23 `std::stacktrace`

- `std::stacktrace` - a dynamic vector, like, of code location
- `stacktrace_entry` - `source_file()`, `source_line()`, `description()` their values are searched in the debugging section, for a given code-address
- `stacktrace::current()` - load the current call sequence stack
- printing value, and formatter - and order of printing
- The `std::stacktrace::current()` uses stack walking to collect the current call sequence
- Similar functionality was available with other libraries, now it is part of the standard
- depends on `'-g'` compiler flag for full details

C++23 std::stacktrace <https://compiler-explorer.com/z/16a1EG7r1>

```
#include <format>
#include <iostream>
#include <print>
#include <stacktrace>
#include <string_view>

void print_current_stacktrace(unsigned indent = 2,
    std::stacktrace trace = std::stacktrace::current()) {
    std::print("=== Stack Trace ({} entries) ===\n", trace.size());
    std::string spaces_((indent+1) * trace.size(), ' ');
    std::string_view spaces(spaces_);
    for (std::size_t i = trace.size() - 1; i < trace.size(); --i) {
        const auto& entry = trace[i];
        auto s = spaces.substr(0, indent * (trace.size() - 1 - i));
        std::println("{}Frame {}: ", s, i);
        std::println("{}Address: {}", s,
            (void*)entry.native_handle());
        std::println("{}Description: {}", s,
            entry.description());
        if (!entry.source_file().empty())
            std::println("{}Location: {}:{}", s,
                entry.source_file(), entry.source_line());
    }
    std::print("=== End Stack Trace ===\n");
}
```

```
void function_c()
{
    std::println("In function_c, printing
stacktrace:");
    print_current_stacktrace();
}

void function_b()
{
    std::println("In function_b, calling
function_c");
    function_c();
}

void function_a()
{
    std::println("In function_a, calling
function_b");
    function_b();
}

int main()
{
    std::println("Starting main, calling
function_a");
    function_a();
}
```

C++23 std::stacktrace <https://compiler-explorer.com/z/a934WfKrP>

```
Starting main, calling function_a
In function_a, calling function_b
In function_b, calling function_c
In function_c, printing stacktrace:
=== Stack Trace (8 entries) ===
Frame 7:
Address: 0xffffffffffffffff
Description:
  Frame 6:
    Address: 0x5df523fc6544
    Description: _start
  Frame 5:
    Address: 0x74e1de029e3f
    Description: __libc_start_main
  Frame 4:
    Address: 0x74e1de029d8f
    Description:
      Frame 3:
        Address: 0x5df523fc6b6c
        Description: main
        Location: /app/example.cpp:49
      Frame 2:
        Address: 0x5df523fc6b2c
        Description: function_a()
        Location: /app/example.cpp:43
      Frame 1:
        Address: 0x5df523fc6aec
        Description: function_b()
        Location: /app/example.cpp:37
      Frame 0:
        Address: 0x5df523fc6a64
        Description: function_c()
        Location: /app/example.cpp:31
    === End Stack Trace ===
```

```
Starting main, calling function_a
In function_a, calling function_b
In function_b, calling function_c
In function_c, printing stacktrace:
=== Stack Trace (7 entries) ===
Frame 6:
Address: 0x7ffb5ff1edcb
Description: ntdll!RtlUserThreadStart+0x2B
  Frame 5:
    Address: 0x7ffb5ec04cb0
    Description: KERNEL32!BaseThreadInitThunk+0x10
  Frame 4:
    Address: 0x7ff6f5f0f410
    Description: output_s!__scrt_common_main_seh+0x10C
    Location: D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl:288
  Frame 3:
    Address: 0x7ff6f5eb61f3
    Description: output_s!main+0x43
    Location: C:\Windows\TEMP\compiler-explorer-compilerQdhW4c\example.cpp:49
  Frame 2:
    Address: 0x7ff6f5eb6193
    Description: output_s!function_a+0x43
    Location: C:\Windows\TEMP\compiler-explorer-compilerQdhW4c\example.cpp:43
  Frame 1:
    Address: 0x7ff6f5eb6133
    Description: output_s!function_b+0x43
    Location: C:\Windows\TEMP\compiler-explorer-compilerQdhW4c\example.cpp:37
  Frame 0:
    Address: 0x7ff6f5eb60c2
    Description: output_s!function_c+0x62
    Location: C:\Windows\TEMP\compiler-explorer-compilerQdhW4c\example.cpp:31
  === End Stack Trace ===
```

Generic Call Context

- `ext::context` a type to capture call sequence information
- `std::stacktrace::current()` - current stacktrace
- `source_location::current()` - instantiation source location
- timestamp `std::chrono::now()`
- thread identification and or name
- process the extra information on construction

ext::context

```
namespace ext {
struct context {
    std::string          msg_;
    std::source_location loc_;
    std::stacktrace      stacktrace_;
    std::chrono::time_point<std::chrono::system_clock> timestamp_;
    std::thread::id      tid_;
    unsigned              throw_count{0};

    context() noexcept {}

    context(const std::string& msg,
            std::source_location loc = std::source_location::current())
        : msg_(msg), loc_(loc), stacktrace_(std::stacktrace::current()),
          timestamp_(std::chrono::system_clock::now()),
          tid_(std::this_thread::get_id()) {}
};
}
```

C++ Exception inherit `std::context`

- `exception : public std::exception, public context {}`
 - Instantiates the exception object on the heap.
 - Provides the call sequences
-
- It requires changes to all thrown exceptions code :-)

Exception handling - Implementation Details

- Linux elf, and the Itanium-ABI
- We need to look into the ABI to understand how we can achieve our goals.
- The `throw x;` statement
 - allocates the exception object on the heap
 - call `__cxa_throw()` function: unwinding the stack and search for a catch block to be activated
 - call `__cxa_begin_catch()` or terminate
- Inject functionality into a function using function interception

Linux - Intercepting a Function

- We need to define the function in our source file
- It needs to be first in the linking order
- We need to find the original function as function pointer
- At the end of our function we can call the original function to continue the execution, as if we weren't there.

Exception - No Code Change at throw Statement

- The internal C++ implementation, the `__cxa_throw`:
 - search for catch block
 - unwinding the stack
 - activate the catch block or abort
- How to inject code for additional processing, while keeping the program running as usual
- Providing extra data, through `thread_local` object from throw to catch
- We can catch the last stacktrace into `thread_local` variable, it will be available at catch block

Intercepting The Exception Handling

- Intercept the `__cxa_throw`:
 - `void __cxa_throw(void *tobj, std::type_info *tinfo, void (dest)(void))` // clang
 - `void __cxa_throw(void *tobj, void *tinfo, void (dest)(void))` // gcc
- Override the default
- Instantiating a `thread_local ext::context`
- communicating with the handler through static variable and `thread_local` for control
- catch blocks have access to the static `thread_local ext::throwinfo` and can call `ext::throwinfo::print()` or `to_string()`
- the terminate handler also has access to the `ext::context` information.

__cxa_throw() interception

```
// Define the signature of the original __cxa_throw function
using __cxa_throw_type = void (*)(void*, std::type_info*,
    void (*)(void*));

// Pointer to store the original __cxa_throw function
static __cxa_throw_type original_cxa_throw {
    (__cxa_throw_type)dlsym(RTLD_NEXT, "__cxa_throw")};
// Our custom __cxa_throw implementation
extern "C" void __cxa_throw(void* thrown_exception,
    std::type_info* tinfo, void (*dest)(void*)) {
    std::println(
        "Intercepted throw called: type: {}", tinfo->name());
    // Call the original __cxa_throw
    original_cxa_throw(thrown_exception, tinfo, dest);
}
```

```
int main() {
    try {
        throw 123;
    }
    catch(...) {
        std::println(
            "got exception")
    }
}
```

output:

```
$ ./minimal throw intercept
Intercepted throw called: type: i
got exception
```

__cxa_throw() interception

```
// https://compiler-explorer.com/z/TnsrvcvWda
#include <dlfcn.h>
#include <print>
#include <typeinfo>

#ifdef __clang__
using __cxa_throw_type = void (*)(void*, std::type_info*, void (*)(void*));
#else
using __cxa_throw_type = void (*)(void*, void*, void (*)(void*));
#endif

// Pointer to store the original __cxa_throw function
static __cxa_throw_type original_cxa_throw((__cxa_throw_type)
dlsym(RTLD_NEXT, "__cxa_throw"));

#ifdef __clang__
extern "C" void __cxa_throw(void* thrown_exception, std::type_info* tinfo, void (*dest)(void*)) {
#else
extern "C" void __cxa_throw(void* thrown_exception, void* vp, void (*dest)(void*)) {
    std::type_info* tinfo = (std::type_info*)vp;
#endif
    std::println("Intercepted throw called: type: {}", tinfo->name());

    // Call the original __cxa_throw function to continue exception handling
    original_cxa_throw(thrown_exception, tinfo, dest);
    abort();
}
```

```
int main()
{
    try {
        throw 123;
    }
    catch(...) {
        std::println("there was an exception");
    }
}
```

output:

```
$ ./minimal_throw_intercept
Intercepted throw called: type: i
there was an exception
```

Compiler Flags

- `-O2` vs `-O0` some of the function call will be inlined
- `-g` - without the debugging information, no filename/line number
- The `LD_PRELOAD` mechanism does not work with statically linked program

Controlling the `ext::throwinfo`

- Capture `ext::context` inside the throw handler.
- Reporting inside the catch block the content using:
`ext::context::to_string()`
- `thread_local` and global variable providing control over the function
- Capture timestamp
- Capture stacktrace
- Callback on throw
- Print/log from throw location
- intercepting the `__cxa_begin_catch`

ext::last_throw_context

// <https://compiler-explorer.com/z/scMh49Tac>

```
namespace ext {
    struct context {
        std::string msg_;
        std::stacktrace stacktrace_;
        std::chrono::time_point<std::chrono::system_clock>
            timestamp_;
        std::thread::id tid_;

        context() noexcept {}
        context(const std::string& msg)
            : msg_(msg),
            stacktrace_(std::stacktrace::current()),
            timestamp_(std::chrono::system_clock::now()),
            tid_(std::this_thread::get_id()) {}
        static context current(std::string msg = "") {
            return context(msg);
        }
    };
    inline thread_local context last_throw_context{};
} // namespace ext
```

```
#ifdef __clang__
extern "C" void __cxa_throw(void* thrown_exception, std::type_info* tinfo, void
(*dest)(void*)) {
#else
extern "C" void __cxa_throw(void* thrown_exception, void* vp, void (*dest)(void*)) {
    std::type_info* tinfo = (std::type_info*)vp;
#endif
    ext::last_throw_context = ext::context::current("exception");
    original_cxa_throw(thrown_exception, tinfo, dest);
    abort();
}

int main() {
    try {
        throw 123;
    }
    catch (...) {
        std::println("there was an exception context:\n{}", ext::last_throw_context);
    }
}

output:
$ ./exception_capture
there was an exception context:
context(msg: exception, @2025-09-08 02:40:24.454096358, 137764763670464) stacktrace:
0# ext::context::context(std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> > const&) at
/home/erez/2025/CppCon2025/throw_stacktrace_prep/slides-examples/35_throw_interception/
exception_capture.cpp:25
1# ext::context::current(std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> > >) at
/home/erez/2025/CppCon2025/throw_stacktrace_prep/slides-examples/35_throw_interception/
exception_capture.cpp:31
2# __cxa_throw at
/home/erez/2025/CppCon2025/throw_stacktrace_prep/slides-examples/35_throw_interception/
exception_capture.cpp:67
3# main at
/home/erez/2025/CppCon2025/throw_stacktrace_prep/slides-examples/35_throw_interception/
exception_capture.cpp:77
4# __libc_start_call_main at ../sysdeps/nptl/libc_start_call_main.h:58
5# __libc_start_main_impl at ../csu/libc-start.c:360
6# _start at :0
7#
```

Terminate Handler

- Printing the last captured exception.
- if multiple threads, print for each one of them, while marking the one without catch / caused the terminate.
- abort()

Creating a Shared Library

- With / without app awareness, using or not using the library API.
- Default behavior to report on terminate which thread killed the process and on segmentation fault - report exceptions it.
- Environment variables - for setting behavior
- Services API:
 - `on_throw_handler`,
 - `last_throw_context`,
 - `per_thread_last_throw`

Exception Handler GDB Debugger Support

- One can set a breakpoint on `__cxa_throw` but sometime it is too noisy.
- How to access the captured stack-trace from within the debugger.
`ext::throwinfo::last_throw_info`
- Call the `ext::throwinfo::print_this_thread_throwinfo()` and `ext::throwinfo::print_all_threads_throwinfo()`

Windows Solution

- <https://compiler-explorer.com/z/3q1rPG85G>
- Structured Exception Handling (SEH)
- Installing Handler -Using the
- printing value, and formatter - and order of printing.

Windows Example:

<https://compiler-explorer.com/z/3q1rPG85G>

```
static thread_local std::stacktrace last_throw_stacktrace;
static std::mutex hook_mutex;
static bool hook_installed = false;

// Windows exception hook using SetUnhandledExceptionFilter
// and vectored exception handling
LONG WINAPI VectoredExceptionHandler(PEXCEPTION_POINTERS
ExceptionInfo)
{
    // Check if this is a C++ exception, MS C++ exception code
    if (ExceptionInfo->ExceptionRecord->ExceptionCode ==
0xE06D7363) {
        last_throw_stacktrace = std::stacktrace::current();
    }
    // Continue with normal exception handling
    return EXCEPTION_CONTINUE_SEARCH;
}
```

```
class StackTraceCapture
{
public:
    StackTraceCapture()
    {
        std::lock_guard<std::mutex>
lock(hook_mutex);
        if (!hook_installed)
        {
            // Install vectored exception
handler
            AddVectoredExceptionHandler(1,
VectoredExceptionHandler);
            hook_installed = true;
        }
    }
};

static StackTraceCapture
stack_trace_capture;
```

C++26 exception `[[with_stacktrace]]`

- No change to the throw location or exception type
- Attribute `[[with_stacktrace]]` at catch block
- very light overhead, as it build the stacktrace as part of the catch block search
- See p2490r3: <https://wg21.link/P2490>

```
try {  
}  
catch ([[with_stacktrace]] std::exception & e) {  
    std::cout << "Exception caught: " << e.what() << "\n";  
    std::cout << "Stacktrace:\n" << std::stacktrace::from_current_exception() <<  
std::endl;  
}
```

Other Stack Tracing Libraries

- Working with other Stack Tracing Libraries
- cpptrace - <https://github.com/jeremy-rifkin/cpptrace>
- backward-cpp - <https://github.com/bombela/backward-cpp>
- boost.Stacktrace - <https://github.com/boostorg/stacktrace>
https://www.boost.org/doc/libs/1_89_0/doc/html/stacktrace.html
- C api backtrace: <https://man7.org/linux/man-pages/man3/backtrace.3.html>

Future Work

- Performance improvements, `stacktrace::current` different allocator.
- Windows platform library
- Additional information capturing
- Signal handler safe capture - segmentation fault as exception?

Summary

- Enrichment of the exception information using `ext::context`
- Passing a `ext::throwinfo` from `throw` statement to `catch` block or `terminate` handler
- Capturing multiple threads' exception information for reporting.
- `LD_PRELOAD` library providing capturing and reporting service
- Windows implementation

My Github Projects

- https://github.com/erez-strauss/lockfree_mpmc_queue - mpmc lock free queue
- https://github.com/erez-strauss/ext_any - ext_eny, variant
- <https://github.com/erez-strauss/hiccups> - hiccups

Enhanced Exception Handling - C++23 `std::stacktrace`

Thank You!