Roberto Gomes Rosmaninho Neto

Bridging Swift Error Handling Model to C++

Belo Horizonte, Minas Gerais

2022

Roberto Gomes Rosmaninho Neto

Bridging Swift Error Handling Model to C++

Final Computing Project Report to the Bachelor Degree in Computer Science of the Universidade Federal de Minas Gerais

Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Supervisor Fernando Magno Quintão Pereira

Belo Horizonte, Minas Gerais

2022

Statement of Authorship

I hereby declare that the thesis submitted is my own work. All direct or indirect sources used are acknowledged as references. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

To my friends and family, the reason of everything.

Acknowledgements

I would like to thank my advisor, Fernando Pereira, who provided me with the best opportunities in my life. My colleagues, Angelica Moreira and Breno Campos, who have always supported me on this path, and my parents, Roberto and Rogeria Rosmaninho, for helping me get this far!

Abstract

Error handling is the process of responding to and recovering from error conditions in the program. In Swift, errors are represented by values of types that conform to the Error protocol. Throwing an error lets you indicate that something unexpected happened, and the normal flow of execution can't continue. A throw statement is used to throw an error. Optional returns are used to represent the absence of a value, but when an operation fails, it's often useful to understand what caused the failure so that code can respond accordingly. Therefore, I propose to bridge the Swift Error Handling modeling to C++ to improve the interoperability between the programming languages. The main idea is to be able to throw a C++ exception that stores a thrown Swift Error that has to be represented by a C++ class. In addition to that, to support C++ programs that don't use exceptions, I propose an additional interoperability mode for throwing functions. When C++ exceptions are disabled, C++ functions should return a result value that contains either the value returned by the function, or the Swift Error value: the Swift::Exception<T> class.

Keywords: Compilers, C++, Swift, std::expected, Error Handling, Exceptions, Programming Languages

List of Program Code

Contents

1 Introduction

Swift is a general-purpose programming language built using a modern approach to safety, performance, and software design patterns. On December 3, 2015, the Swift language, supporting libraries, debugger, and package manager were published under the Apache 2.0 license with a Runtime Library Exception. The source code is hosted on GitHub where it is easy for anyone to get the code, build it themselves, and even create pull requests to contribute code back to the project. (Swift.org, 2022c) The effort to improve and optimize the Swift Compiler has been one of the main goals of the Swift community, mainly discussing ideas, goals, implementations, and implementation on specific topics at the Swift Online Fórum.

The growing number of APIs and projects using the Swift Programming Language has been attracting attention not only from the Swift community but also from the C++ community as well. Therefore, if anyone from the C++ community wants to use some API developed in Swift or if they want to incrementally change the codebase of their application from C++ to Swift, they should have the ability to do that by relying on interoperability between the two languages. That's the reason that makes C++ interoperability with Swift so important.

In order to use Swift APIs or incrementally adopt Swift in a C++ code, developers must ensure that their application will not crash due the errors from C++ code or errors from Swift code. Handling errors from C++ in C++ code is straightforward. On the other hand, if a Swift function used in a C++ codebase through a bridging header throws an Error, it's currently impossible to know what happened from the C++ side. So, the program cannot handle errors from Swift and may crash without any meaningful message.

This project aimed to develop and implement the infrastructure on the Swift Compiler to extend the Swift handling errors to C++ through a bridging header.

1.1 General Goals

This project aimed to extend the Swift compiler to allow Swift functions to propagate errors to C++ code. The first modification was to modify the current implementation to enable casting the "Swift::Error" C++ Class to the user's defined errors. The second modification provided another C++ class that also wraps the Swift Error but returns it wrapped in a *std::expected*(cplusplus.com, 2021) like implementation to be used by users that disable the C++ exceptions to handle the error in their way instead of throwing it.

1.2 Specific Goals

The goals of this projects can me summarized in:

- Create a new C++ class, like *std::exception*, and stores a Swift error value.
- Extend this C++ Swift Error representation to cast to the user's defined errors.
- Generate a C++ class similar to *std::expected*(cplusplus.com, 2021) to provide error handling for clients that don't use C++ exceptions.
- Automatically switch the error handling model regarding the cpp_exceptions flag value.

2 Theoretical References

2.1 History

The history of the C++ programming language starts in 1979, when Bjarne Stroustrup was doing work for his Ph.D. thesis. He began working on "C with Classes", which as the name implies was meant to be a superset of the C language. His goal was to add object-oriented programming into the C language. His language included classes, basic inheritance, inlining, default function arguments, and strong type checking in addition to all the features of the C language. (Albatross, 2021)

The first C with Classes compiler was called Cfront, which was derived from a C compiler called CPre. It was a program designed to translate C with Classes code to ordinary C. Cfront would later be abandoned in 1993 after it became difficult to integrate new features into it, namely C++ exceptions. (Albatross, 2021)

2.2 Exceptions

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called handlers.

To catch exceptions, a portion of the code is placed under exception inspection. This is done by enclosing that portion of code in a try-block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally, and all handlers are ignored. (cplusplus.com, 2021)

An exception is thrown by using the throw keyword from inside the ***try*** block. Exception handlers are declared with the keyword ***catch***, which must be placed immediately after the try block:

```cpp
// exceptions
#include <iostream>
using namespace std;

int main () {
  try {
    throw 20;
  } catch (int e) {
    cout << "An exception occurred. Exception Nr. " << e << '\n';
  }
  return 0;
}
```

Listing 1 – Example of the Exception Handling in C++.

The code under exception handling is enclosed in a try block. In this example, this code simply throws an integer number as an exception:

```cpp
throw 20;
```

Listing 2 – Example of a simple throw statement.

2.3 Swift Error Handling Model

In Swift, errors are represented by values of types that conform to the Error protocol. Throwing an error indicates that something unexpected happened, and the normal flow of execution can't continue.

When a function encounters an error condition, it throws an error. That function's caller can then catch the error and respond appropriately. (Swift.org, 2022a)

```swift
enum numError: Error {
    case twenty
}

func main () throws -> Int {
    do {
        throw numError.twenty
    } catch {
        print("An exception occurred. Exception Nr. \(error)");
    }
    return 0;
}
```

Listing 3 – Example of declaration of a function that can throw an error. This is almost an equivalent program to 1.

A function indicates that it can throw an error by including the ***throws*** keyword in its declaration. The keyword ***throw*** was used in C++ as well, at least until the C++17 standard.(cppreference.com, 2022) However, unlike Swift, in C++, this keyword works as a dynamic exception specification followed by a type indicating that the function with it may throw exceptions of that type or a type derived from it. On the other hand, if the function is followed by ***throw()***, the compiler knows that it can't throw an exception.

When you call a function that can throw an error, you prepend the try keyword to the expression. In Swift, this structure is called ***do-try-catch***. This structure is very common in codebases that choose to work with exceptions.

```swift
do {
    try canThrowAnError()
    // no error was thrown
} catch {
    // an error was thrown
}
```

Listing 4 – Example of the correct implementation to call a function that can throw an error.

If a function throws an error, then Swift automatically propagates this error out of its current scope until a catch clause handles it, even if this function is called from a C++ program. That is one of the reasons that motivate this work to improve the interoperation between Swift and C++ by creating two solutions to handle Swift Errors in C++.

2.4 The std::expected<T,E>

The proposal p0323r3(Botet, 2017b) presents the class template ***expected<T, E>*** that contains either:

- A value of type T, the expected value type; or
- A value of type E, an error type used when an unexpected outcome occurred.

The interface can be seen as whether the underlying value is the expected value (of type T) or an unexpected value (of type E). The original idea comes from Andrei Alexandrescu C++ and Beyond 2012: Systematic Error Handling in C++ Alexandrescu(Alexandrescu, 2012). The interface and the rationale were based on ***std::optional*** (Cacciola, 2013). The proposal's author considers ***expected<T, E>*** as a supplement to ***optional<T>***, expressing why an expected value isn't contained in the object.

It's important to recap that C++'s two main error mechanisms are exceptions and return codes. Characteristics of a good error mechanism are(Botet, 2017b):

1. Error visibility: Failure cases should appear throughout the code review: debugging can be painful if errors are hidden.
2. Information on errors: Errors should carry information from their origin, causes, and possibly the ways to resolve them.
3. Clean code: Treatment of errors should be in a separate layer of code and as invisible as possible. The reader could notice the presence of exceptional cases without needing to stop reading.
4. Non-Intrusive error Errors should not monopolize the communication channel dedicated to normal code flow. They must be as discrete as possible. For instance, the return of a function is a channel that should not be exclusively reserved for errors.

The same analysis can be done for the ***expected<T, E>*** class and observe the advantages over the classic error reporting systems.(Botet, 2017b)

1. Error visibility: It takes the best of the exception and error code. It's visible because the return type is ***expected<T, E>***, and users cannot ignore the error case if they want to retrieve the contained value.
2. Information: Arbitrarily rich.
3. Clean code: The monadic interface of expected provides a framework delegating the error handling to another layer of code. Note that ***expected<T, E>*** can also act as a bridge between an exception-oriented code and a nothrow world.

4. Non-Intrusive Use the return channel without monopolizing it.

3 Methodology

3.1 Current State of the C++ and Swift Interoperability

The interoperability between Swift and C++ programming languages works through a bridging header. This header is a C++ file containing all the function signatures, types, and class declarations that a user wants to export from a Swift source code to a C++ program. To generate this header, we must modify the Swift Compiler to correctly recognize and represent the information we want to export using C++ types, macros, functions, and other useful resources.

3.2 Initial implementation

The main goal of this work is to continue translating the Swift Error Handling to C++ Error Handling.

To achieve this goal, I got in touch with Alex Lorenz, the Apple employee assigned to work with students in the Swift Interoperability Work-group(Swift.org, 2022b). The group is composed of community members and Apple employees from the Clang team. We did sync-up meetings every week to ask questions about any blockers we had on our projects, to know each one and the projects better, and to get guidelines from Apple's perspective. Also, the Work-group has a Slack channel public where I continued in touch with Alex to ask simple implementation questions and to discuss about GitHub CI tests and reviews.

The first step was to study Swift and C++ handling error model individually from the user's perspective, write the example 1 using both programming languages and analyze the syntax of each one, as shown in the Theoretical References section. Then, I started studying the Swift Compiler, especially the Code Generation part on the PrintAsClang(Swift, 2022) directory.

The second step, and my first official contribution(Swift#59217, 2022), following a Test-Driven Development, was to create tests with functions that throw an error. The idea was to implement a simple Enum Error with two cases and two functions: the first marked with "throws" flag but never throwing any error, and the second with "throws" as well, but throwing an error from the Enum error created earlier.

The third step, and my second official contribution(Swift#59787, 2022), was to modify the compiler to delete all flags that classify every function as a "noexcept" function. In the previous implementation and from a C++ point of view, no function could throw

errors. The solution to solve this hardcoded behavior was to verify during the header code generation if a function was declared in Swift with the flag "throws" or not. This modification made the tests on (Swift#59217, 2022) pass as expected, as they were functions that could throw in Swift, but were marked as "noexcept" in the C++ bridge header.

Unfortunately, this last step was not straightforward as expected. The "GitHub CI" pointed out that one single test from the entire Test Suit failed on the architecture "i386-apple-watchos2.0-simulator". After some debugging, we realized that two default parameters were missing on the function signature, the first called "self" and the second called "error". The former was crucial to the next steps as it contains the information of an error if one was thrown during the execution of the function.

```
1  inline void throwFunction() {
2    void* opaqueError = nullptr;
3    void* self = nullptr;
4    _impl::$s9Functions13throwFunctionyyKF(self, &opaqueError);
5    if (opaqueError != nullptr)
6      throw (swift::_impl::NaiveException("Exception"));
7  }
```

Listing 5 – Example of a function that can throw a NaiveException if the Swift function correspondent throws an error.

The above example illustrates the third contribution(Swift#60079, 2022) of this project:

- The "if condition"testing if the "opaqueError" was modified during the execution of the Swift function.
- The NaiveException C++ class.

The first modification tests if the Swift function throws an error. In this case, the "opaqueError" parameter holds the information needed to identify the thrown error instead of "nullptr". The second modification implements a C++ class to construct a simple object with a message and throw it to the C++ caller function. Also, this message can be accessed with the "***getMessage()***" method.

3.3 Final Implementation

The first part of this project aimed to provide a minimal version of the Swift handling error model to C++, so a C++ function caller could know that a Swift function called throws an error. The second and final part of this project aimed to specialize that knowledge by providing a general "***Swift::Error***" representation, an infrastructure to recognize which error case the function threw, and a new C++ class ***Swift::Expected***" to return the error thrown.

The fourth contribution(Swift#60858, 2022) of this project was the mentioned general "***Swift::Error***" representation to replace the "NaiveException" on the "throw" statement of the C++ function representation in the generated bridging header. This class implements its constructor and destructor using the functions "***swift_errorRetain***" and "***swift_errorRelease***" to handle the Swift Error properly avoiding unexpected behaviors and memory leaks. The main idea of this class was to provide a representation for the Swift error type that can be examined on the C++ side. This representation behaves as Swift protocol type in C++, and is able to represent all Swift error values in C++.

The main idea of this class was to replace the need for using the "std::exception" library, that currently has dependencies that can't be imported into the generated bridging header, and to work as a "superclass" for all Swift Error in C++.

```
1  class TestDestroyed {
2    deinit {
3      print("Test destroyed")
4    }
5  }
6
7  public struct DestroyedError : Error {
8    let t = TestDestroyed()
9  }
10
11 public func testCast(_ e: Error) {
12     let _ = e as? DestroyedError
13 }
```

Listing 6 – Swift function that can throw an Error, but doesn't have a throw statement.

The fifth contribution(Swift#61626, 2022) was the most challenging part of this project. It required debugging the example 6 at the assembly level using the lldb(Debugger, 2007) trace and breakpoints to discover the information needed to reproduce the casting at line 12 using our C++ infrastructure.

In a 1-1 debugging meeting with Alex from Apple, we found out that to dynamically

cast an error we need some metadata information, a symbolic type, and external global variables. Some of these required pieces of information could be hardcoded, and others we could get by declaring them as "external "C"" functions and variables since the Swift Compiler uses them to build the LLVM IR.

Finally, after implementing all the auxiliary functions to dynamically cast the "***Swift::Error***" to a concrete Swift type that represents the case thrown, I implemented the "***as<T>()***" function as a method from the "***Swift::Error***" class to provide the user a natural experience to handle errors in C++.

In the sixth contribution (Swift#62197, 2022), I improved the mentioned dynamic cast to return a "***Swift::Optional<T>***" type within a value if the casting was successful or a "***Swift::Optional<T>::none()***" if not. Also, I deleted the "***NaiveException***" that wasn't necessary anymore and changed "***Swift:Error***" implementation to the same file where "***Swift::Optional<T>***" was defined to be able to modify its return type.

Finally, in the seventh and last contribution(Swift#61823, 2022), I developed and implemented my own version of the "***std::Expected<T>***"(Botet, 2017a) proposal as "***Swift::Expected<T>***" and implemented the support for the bridging header chose the proper way to handle a Swift thrown error according to the C++ user preference to use or not exceptions by passing the "*-fno-exceptions*" flag to the compiler. The first part of this project was very challenging due to the responsibility to build a great API that will be used extensively by the community. Also, this is the implementation of a proposal that has been discussed for years in the community, with only a few people risking implementing their version of "***std::Expected<T>***".

Also, two important requirements from Apple were to use the "***Swift:Error***" as the only type of error that could be returned and to store at the same buffer the error or the value of type T to optimize the memory for objects from this class. The last part of this contribution wasn't trivial either: in order to avoid code repetition either on the compiler and on the bridging header, I used an alias (*ThrowingResult*) and a Macro *(SWIFT_RETURN_THUNK)* to specify which implementation of the part of the C++ implementation would be used to properly represent the Swift function regarding the error modeling choose by the C++ user as shown below:

```
#ifdef __cpp_exceptions
 template<class T>
 using ThrowingResult = T;
 #define SWIFT_RETURN_THUNK(T, v) v
 #else
 template<class T>
 using ThrowingResult = Swift::Expected<T>;
 #define SWIFT_RETURN_THUNK(T, v) Swift::Expected<T>(v)
 #endif
```

Listing 7 – Macro definition to replace the function type and return regarding the use of exceptions.

4 Results and Example

The contributions cited above show that we successfully accomplished our primary goals. Finally, using two approaches, C++ users can handle errors thrown by Swift functions. Below, the straightforward example shows a Swift function that can throw an error to avoid unexpected behavior.

```
@_expose(Cxx)
public enum DivByZero : Error {
    case divisorIsZero
    case bothAreZero

    // Function to print the case thrown
    public func getMessage() {
        print(self)
    }
}

@_expose(Cxx)
public func division(_ a: Int, _ b: Int) throws -> Float {
    if a == 0 && b == 0 {
        throw DivByZero.bothAreZero
    } else if b == 0 {
        throw DivByZero.divisorIsZero
    } else {
        return Float(a / b)
    }
}
```

Listing 8 – Example of a user-defined Error and a Swift function that can throw it.

The client of this function, the C++ user, must first compile the program in which this function is defined to create the bridging header to export the Swift implementation to C++. The below command can be used to generate this header; it was extracted from the Swift test suit and simplified as much as possible.

```
$ swift-frontend Example.swift -typecheck -module-name Functions \
    -enable-experimental-cxx-interop -emit-clang-header-path functions.h
```

Listing 9 – Command to create the bridging header of a Swift program.

This command takes the "Example.swift" file as input and outputs the "functions.h" file with the C++ representation of the Swift functions under the "Functions" namespace.

```
1  inline Swift::ThrowingResult<float> division(swift::Int a, swift::Int b) {
2    void* opaqueError = nullptr;
3    void* _ctx = nullptr;
4    auto returnValue = _impl::$s9Functions8divisionySfSi_SitKF(a,b,_ctx,&opaqueError);
5    if (opaqueError != nullptr)
6  #ifdef __cpp_exceptions
7      throw (Swift::Error(opaqueError));
8  #else
9      return SWIFT_RETURN_THUNK(float, Swift::Error(opaqueError));
10 #endif
11
12   return SWIFT_RETURN_THUNK(float, returnValue);
13 }
```

Listing 10 – Example of using a Swift function that can throw an Error in C++ that can be caught.

The C++ function represented generated by the compiler, and shown above, can be explained in four parts:

- **Function signature**: The function has the type "***ThrowingResult<float>***" instead of the "***float***" type declared on Swift. The implementation of this type is shown on listing 7, and it is used to avoid code repeating by declaring the function twice with different types to satisfy the two error handling approaches.
- **The Swift function call**: This has two extra arguments we defined before; the first holds the context, and the second the error information. They are required to match the Swift ABI, and the last argument has the core information to specify if the Swift function threw an error and which error it is or if it returned as expected.
- **The opaqueError is nullptr?**: This if condition checks if the "***opaqueError***" argument was modified during the execution of the function to hold Swift error information or not. Also, inside the branch, we have two different implementations regarding the use of exceptions by the C++ user; if it's enabled, we throw a "***Swift::Error***" object initiated with the "***opaqueError***" argument. If not, we **return** a macro that will be replaced by "***Swift::Expected<float>(Swift::Error(opaqueError))***".
- **The return expression**: The macro returned here will be replaced at compile time depending on the use of C++ exceptions. It can either only returns the "***returnValue***" or the "***Swift::Expected<float>(returnValue)***" that contains the value we expected instead of an Error as it previous use in this example.

The C++ programmer uses this function representation by importing the generated Swift bridging header to its program. Below we show two uses of this function using C++ exceptions:

```cpp
#include <cassert>
#include <cstdio>
#include "functions.h"

int main() {

    // This example catches an exception
    try {
        auto result = Functions::division(0,0);
        printf("result = %f\n", result);
    } catch (Swift::Error& e) {
        auto errorOpt = e.as<Functions::DivByZero>();
        assert(errorOpt.isSome());

        auto errorVal = errorOpt.get();
        assert(errorVal == Functions::DivByZero::bothAreZero);
        errorVal.getMessage();
    }

    // This example gets the correct value returned by the function
    try {
        float result = Functions::division(4,2);
        printf("result = %f\n", result);
    } catch (Swift::Error& e) {
        auto errorOpt = e.as<Functions::DivByZero>();
        assert(errorOpt.isSome());

        auto errorVal = errorOpt.get();
        errorVal.getMessage();
    }
    return 0;
}
```

Listing 11 – Examples of using a Swift function that can throw an error in C++ and handles it.

In this example, we can see how to catch the "***Swift::Error***", dynamically cast it to the user-defined error, which returns an Optional type that says if the casting was successful or not, check if it threw the case that we expected, and then, tells the user name of it using the "***getMessage()***" function. If the function doesn't throw an error, we get the "***float***" result and print it to the standard output.

The commands below show how to compile and test this program. First, we need to compile and generate its object file, compile it with the Swift file containing the function and error definitions, and then generate the executable program.

```
$ clang++ -I ${SWIFT_BUILD_DIR}/swift-macosx-arm64/./lib/swift \
        -c Example.cpp -I /. -o example.o

$ swiftc -Xfrontend -enable-experimental-cxx-interop Example.swift -o example \
        -Xlinker example.o -module-name Functions -Xfrontend \
        -entry-point-function-name -Xfrontend swiftMain

$ ./example

```

Listing 12 – Commands to build, link and execute a C++ program that calls a Swift function and can throw exceptions

The second approach to handle Swift errors in C++ was inspired by the C++ proposal "p0323r4 std::expected"(Botet, 2017a). This approach required a new C++ parameterized class called "Swift::Expected<T>", which can be initialized to hold either a Swift::Error or a value of type T, never simultaneously. This class has auxiliary functions to improve the interface with the user and simplify the test and get an error or a value. An instance of this class is returned by a C++ representation of a Swift function that can throw an error, but the C++ user disabled exceptions at compile time.

The example below uses the same Swift function and C++ representation of it but with this new error-handling model:

```cpp
#include <cassert>
#include <cstdio>
#include "functions.h"

int main() {
    auto errorResult = Functions::division(1,0);
    if (errorResult.has_value()) {
        printf("result = %f\n", errorResult.value());
    } else {
        auto optionalError = errorResult.error().as<Functions::DivByZero>();
        assert(optionalError.isSome());

        auto errorValue = optionalError.get();
        assert(errorValue == Functions::DivByZero::divisorIsZero);
        errorValue.getMessage();
    }

    auto goodResult = Functions::division(4,2);
    if (goodResult.has_value()) {
        printf("result = %f\n", goodResult.value());
    } else {
        auto optionalError = goodResult.error().as<Functions::DivByZero>();
        assert(optionalError.isSome());

        auto errorValue = optionalError.get();
        errorValue.getMessage();
    }
    return 0;
}
```

Listing 13 – Handling Swift Error using an expected type based on the std::expected proposal.

The main difference in this example is that the “try-catch” block is not used. Instead, the function’s return is tested to check if it contains a value, the good case where we just print the value or not. This last case is most interesting, as the error was returned and can be accessed by the “error()” function and then handled as before.

The commands below show how to compile and test this program. First, we need to compile and generate its object file with the “-fno-exceptions” flag, compile it with the Swift file containing the function and error definitions and then generate the executable program.

```
$ clang++ -I ${SWIFT_BUILD_DIR}/swift-macosx-arm64/./lib/swift \
        -c Expected_Example.cpp -fno-exceptions -I /. -o expected_example.o

$ swiftc -Xfrontend -enable-experimental-cxx-interop Example.swift \
        -o expected_example -Xlinker expected_example.o -module-name Functions \
        -Xfrontend -entry-point-function-name -Xfrontend swiftMain

$ ./expected_example
```

Listing 14 – Commands to build, link, and execute a C++ program that calls a Swift function and can't throw exceptions, instead, it uses the Swift::Expected<T> class.

References

Albatross. *History of C++*. 2021. Last accessed 6 May 2022.

Alexandrescu, A. *Systematic Error Handling in C++*. 2012. Last accessed 19 Dez 2022. Available on: <https://isocpp.org/blog/2012/12/systematic-error-handling-in-c-andrei-alexandrescu>.

Botet, J. B. V. *p0323r4 std::expected*. 2017. Last accessed 5 May 2022. Available on: <http://wg21.link/P0323r4>.

Botet, J. B. V. *Utility class to represent expected object*. 2017. Last accessed 19 Dez 2022. Available on: <https://wg21.link/p0323r3>.

Cacciola, A. K. F. *A proposal to add a utility class to represent optional objects*. 2013. Last accessed 19 Dez 2022. Available on: <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3793.html>.

cplusplus.com. *Exceptions - C++ Tutorials*. 2021. Last accessed 6 May 2022. Available on: <https://www.cplusplus.com/doc/tutorial/exceptions/>.

cppreference.com. *Dynamic exception specification (until C++17)*. 2022. Last accessed 20 Jul 2022. Available on: <https://en.cppreference.com/w/cpp/language/except_spec>.

Debugger, T. L. *The LLDB Debugger*. 2007. Last accessed 18 Dez 2022. Available on: <https://lldb.llvm.org>.

Swift. *Swift Programming Language*. 2022. Last accessed 18 Jul 2022. Available on: <https://github.com/apple/swift/tree/main/lib/PrintAsClang>.

Swift#59217. *[Interop] [SwiftToCxx] New Throw Error Test #59217*. 2022. Last accessed 18 Jul 2022. Available on: <https://github.com/apple/swift/pull/59217>.

Swift#59787. *[Interop] [SwiftToCxx] New Throw Error Test #59217*. 2022. Last accessed 18 Jul 2022. Available on: <https://github.com/apple/swift/pull/59787>.

Swift#60079. *[Interop] [SwiftToCxx] Implementing a naive exception to be thrown in C++ if an Error is thrown in Swift #60079*. 2022. Last accessed 18 Jul 2022. Available on: <https://github.com/apple/swift/pull/60079>.

Swift#60858. *[SwiftToCxx] Including Cxx representation of Swift's Error #60858*. 2022. Last accessed 18 Dez 2022. Available on: <https://github.com/apple/swift/pull/60858>.

Swift#61626. *[Interop] [SwiftToCxx] Introduce Dynamic Cast to swift::Error #61626*. 2022. Last accessed 18 Dez 2022. Available on: <https://github.com/apple/swift/pull/61626>.

Swift#61823. *[Interop][SwiftToCxx] Introduces swift::Expected #61823*. 2022. Last accessed 19 Dez 2022. Available on: <https://github.com/apple/swift/pull/61823>.

Swift#62197. *[Interop] [SwiftToCxx] Modify the swift::Error Dynamic Cast to return a Swift::Optional #62197*. 2022. Last accessed 18 Dez 2022. Available on: <https://github.com/apple/swift/pull/62197>.

Swift.org. *The Basics - The Swift Programming Language (Swift 5.6)*. 2022. Last accessed 6 May 2022. Available on: <https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID335>.

Swift.org. *C++ Interoperability Workgroup*. 2022. Last accessed 11 Jul 2022. Available on: <https://www.swift.org/cxx-interop-workgroup/>.

Swift.org. *Swift.org - Welcome to Swift.org*. 2022. Last accessed 5 May 2022. Available on: <https://www.swift.org>.