SlimExc

Documentation

A deterministic exception handling implementation for C++ with GCC

Documentation version: 0.9.0
for SlimExc-plugin version: 0.9.0
and SlimExc-library version: 0.9.0

Philipp Rimmele - developer@philipp-rimmele.de Valentin Felder - fritz-valentin@web.de

June 16, 2024

Contents

1. Motivation

1.1 Status quo

The C++ standard exception handling implementation is not deterministic in the sense that no upper bound for the runtime and memory usage can be statically calculated [1, chapter 2.5]. Also the error path has a significantly increased runtime compared to the success path. Beyond that the initial size of the library which implements the necessary functionalities is not negligible on smaller systems. section 5
This makes it unsuitable for systems which are either time-, memory-, or safety-critical. For this reason the exception handling gets banned partially or completely in over 50% of all C++ projects [1, chapter 2.1]. Many projects resort to using other means of error handling, which can only ever represent a non-standard dialect of C++ and come with their own drawbacks [1, chapter 3.1].

1.2 The goal of SlimExc

SlimExc aims to provide an alternative implementation of the C++ exception handling mechanisms, which is standard-conform to a high degree (Deviations from the standard are listed in subsection 4.3), while avoiding many of the issues of the standard implementation. Mainly, SlimExc

- is runtime deterministic
- is memory deterministic
- has a minimal library overhead
- has a comparable runtime on both the error- and success paths
- has the same syntax and semantics as standard try-catch exception handling
- supports dynamically typed exception objects
- integrates seamlessly into the GNU Compiler Collection (GCC) toolchain

2. Working principle

2.1 Return based approach

Every time an exception is thrown, the normal program execution must be interrupted and continued at the appropriate catch site. Therefore the stack must be unwound in some way until the the stackframe of the catch site is reached. In the C++ standard exception handling implementation this is done with a special stack-unwinding routine, which also takes care of the destructor calls for all objects on the stack. This routine is very runtime intensive as shown in subsection 5.2. More details about the standard implementation can be found at [2].
The Slim-Exception-Implementation instead uses return statements to unwind the stack. The return semantics also imply that all destructors for objects on the stack are called. A schematic overview of the different control flows is shown in Figure 1 and Figure 2.

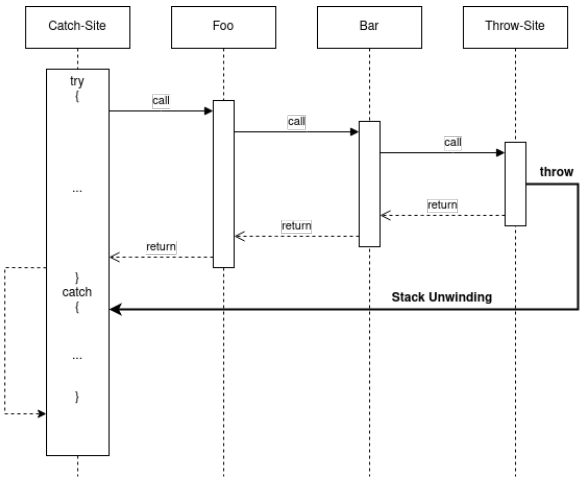

Figure 1: Control flow of default exception handling

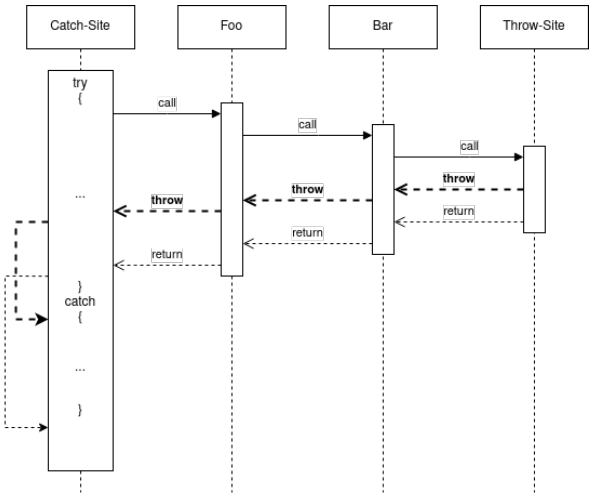

Figure 2: Control flow of Slim-Exception-handling

In the standard implementation the success and error paths are fully distinct. Therefore there is no effective runtime overhead in the success path, but there is a significant non-deterministic runtime overhead in the error path. In the return based approach the same return mechanism is used for the error and success paths. To distinguish between error and success, additional checks must be inserted. This only leads to a small deterministic runtime overhead for both paths.

2.2 Exception storage on the stack

In the C++ standard exception handling implementation the exception object is either stored on the Heap (dynamic memory) or on the stack of the throw-site (stack-pinning). Both, stack-pinning and dynamic memory allocation, are not space- and/or time deterministic. More details about the standard implementation can be found at [2].

The Slim-Exception-Implementation instead stores the exception object on the stack of the catch site. Therefore, no stack-pinning is needed. To realize this mechanism it is necessary to allocate the memory of the largest possible exception object already when a try-block is entered. This leads to a constant (deterministic) memory overhead for both the success and error paths.

2.3 Runtime and memory behavior

The focus of the C++ standard exception handling implementation is to have a minimal runtime overhead in the success path, which is achieved with a rather significant non-deterministic runtime overhead in the (exceptional) error path. The focus of the Slim-Exception-Implementation is to have a deterministic space and runtime overhead in both paths.

The exception handling library of the standard implementation also results in a significant program-size overhead if any exceptions are used. The Slim-Exception-Implementation has a minimal library size, but adds a small additional program-size overhead for each usage of exception handling mechanisms.

A comparison between the overheads of the different implementations is made in section 5.

3. Implementation

3.1 Slim-Exception-Library

The core functionality of the Slim-Exception-System is implemented in the class "ExceptionState" within the Slim-Exception-Library. A instance of this class exists for each try-catch-block. Whenever possible the methods are implemented as inline functions to minimize the effective runtime. An overview of the class declaration is shown in Listing 1.

```
1  namespace SlimExcLib
2  {
3  class ExceptionState final
4  {
5  private:
6     unsigned char exceptionBuffer[__SLIM_EXC_BUFFER_SIZE] alignas(alignof(std::max_align_t));
7     InstanceType typeId;
8     void(*destruct)(const void*) noexcept;
9     ExceptionState* previousES = NULL;
10
11    enum State : uint8_t
12    {
13       CLEAR = 0,
14       HANDLERETHROW = 1,
15       HANDLETHROW = 2,
16       THROW = 3,
17       RETHROW = 4
18    } state = CLEAR;
19
20
21    template <class T> static inline bool compareAdresses(T& exception, void* bufferAdr) noexcept;
22
23    template<class T> static void destructorInvoker(const void* obj) noexcept;
24
25    void propagateUp() noexcept;
26    void takeInstance(ExceptionState* source) noexcept;
27    inline ExceptionState* getLatestHandlingExceptionState(void)noexcept;
28    template <class T> bool throwExceptionHelper(T& exc) noexcept;
29
30 public:
31    ExceptionState(ExceptionState* previous) noexcept;
32    ~ExceptionState() noexcept;
33
34    static ExceptionState* getCurrentExceptionState() noexcept;
35    static void setCurrentExceptionState(ExceptionState* newInstance) noexcept;
36
37
38    inline bool isExceptionInState(State state) noexcept;
39    inline bool isExceptionThrowing() noexcept;
40    inline void setToHandlingState() noexcept;
41    inline void setToThrowingState() noexcept;
42    inline void setToHandleRethrowState() noexcept;
43    inline void setToRethrowingState() noexcept;
44
45    inline void rethrow(void) noexcept;
46    template <class T> inline bool holdsExceptionOfTypeT() noexcept;
```

```
47      template <class T> inline void* getExceptionReference() noexcept;
48      template <class T> inline void throwException(T& exc) noexcept;
49      template <class T> inline void throwException(T&& exc) noexcept;
50  };
51  }
```

Listing 1: Declaration of the class "ExceptionState"

Attributes:

- **exceptionBuffer:** The raw buffer where the exception object gets stored. The size "__SLIM_EXC_BUFFER_SIZE" is set by the Slim-Exception-Plugin and can be defined in the configuration.
- **typeId:** The type of the currently stored exception. The datatype of this attribute depends on the selected RTTI-Implementation. This can also be defined in the configuration.
- **destruct:** A pointer to the destructor of the currently stored exception.
- **previousES:** A pointer to the "ExceptionState"-instance of the next outer try-catch-block. All "ExceptionState"-instances form a linked list which is required for rethrowing and exception propagation.
- **state:** An enumeration which defines the current Exception state. Listing 2 shows an example for each state. Possible values are:
  - **CLEAR (line 1):** Initial state. No exception is currently thrown or handled in this "ExceptionState"-instance. A new exception can be thrown in this state.
  - **THROW (line 3 & 15):** After a new exception is thrown the current "ExceptionState"-instance is set to this state. It remains in this state until the exception is caught. This state is also used for rethrowing if the current "ExceptionState"-instance holds the exception to rethrow. In this case the current "ExceptionState"-instance get set from "HANDLETHROW" to "THROW" again.
  - **HANDLETHROW (line 5):** After a new exception (current "ExceptionState"-instance in state "THROW") is caught, the current "ExceptionState"-instance is set to this state. It remains in this state until the handling catch-Block is exited (by return, rethrow, new throw or by leaving the block after reaching its end).
  - **RETHROW (line 9):** This state is only used if a rethrow occurs, but the exception to rethrow is not in the current "ExceptionState"-instance. In this case the linked list of "ExceptionState"-instances is traversed to check if an instance lower down the list is in the state "HANDLETHROW", and if so, the current "ExceptionState"-instance is set to state "RETHROW". Otherwise "std::terminate" will be called.
  - **HANDLERETHROW (line 11):** After a rethrown exception (current "ExceptionState"-instance in state "HANDLERETHROW") is caught, the current "ExceptionState"-instance is set to this state. The exception object will be retrieved from the "ExceptionState"-instance lower down the linked list, which is in state "HANDLETHROW" and thus contains the rethrown exception object. It remains in this state until the handling catch-Block is exited (by return, rethrow, new throw or by leaving the block after reaching its end).

```
1  try //Creates "ExceptionState"-Instance 1 (ES1) (ES1.state = CLEAR)
2  {
3     throw 200; //ES1.state = THROW, Buffer holds int(200)
4  }
5  catch(int& e) //ES1.state = HANDLETHROW
6  {
7     try //Creates "ExceptionState"-Instance 2 (ES2) (ES2.state = CLEAR)
8     {
```

```
9         throw;  //rethrow "e". ES2.state = RETHROW
10    }
11    catch(int& e) //ES2.state = HANDLERETHROW
12    {
13    }
14
15    throw; //rethrow "e". ES1.state = THROW (Not RETHROW!)
16 }
```

Listing 2: Example code to show the usage of all the states

Methods:

- **compareAdresses:** (helper) Compares the addresses between a given Exception object and the buffer of this "Exception-State" instance. This is needed in case an explicit rethrow occurs within a catch-by-reference block, in order to prevent copying an exception object into itself.
- **destructorInvoker:** (helper) Template function to invoke a destructor call on an anonymous object interpreted as object of type T. Gets used in combination with the "destruct"-attribute to correctly destroy the exception object in the "exceptionBuffer" of this "ExceptionState"-instance.
- **propagateUp:** (helper) This method gets used when the lifetime of this "ExceptionState"-instance ends, but it contains an active exception object which needs to be preserved in order to propagate up the callstack. This method moves the current "ExceptionState"-instance into the previous instance (following the linked list). In case the previous "ExceptionState"-instance is already in the state "THROWING", or there is no previous "ExceptionState"-instance, "std::terminate" gets called.
- **takeInstance:** (helper) Sets the internal state of this "ExceptionState"-instance to the one of another given "ExceptionState"-instance. Gets used by "propagateUp".
- **getLatestHandlingExceptionState:** (helper) Returns the most recent "ExceptionState"-instance which is in state "HANDLETHROWING" from the linked list, excluding the current "ExceptionState"-instance. Gets used to deal with states "RETHROW" and "HANDLERETHROW".
- **throwExceptionHelper:** (helper) Used to set up the internal state of this "ExceptionState"-instance to hold a new exception-Object. Returns "true" on success.
- **ExceptionState:** Constructor. Every time a new "ExceptionState"-instance is created, it adds itself to the linked list and sets itself as the new current "ExceptionState"-instance by calling "setCurrentExceptionState".
- **~ExceptionState:** Destructor. When the lifetime of this "ExceptionState"-instance ends, the previous "ExceptionState"-instance gets set as current "ExceptionState"-instance again by calling "setCurrentExceptionState". If this "ExceptionState"-instance is in state "THROWING" or "RETHROWING", "propagateUp" gets called.
- **getCurrentExceptionState:** Implementation is up to the user, see subsubsection 4.1.4.
- **setCurrentExceptionState:** Implementation is up to the user, see subsubsection 4.1.4.
- **isExceptionInState:** Checks if this "ExceptionState"-instance is in a given state.
- **isExceptionThrowing:** Checks if this "ExceptionState"-instance is in state "THROWING" or "RETHROWING".
- **setToHandlingState:** Sets this "ExceptionState"-instance to state "HANDLETHROW".
- **setToThrowingState:** Sets this "ExceptionState"-instance to state "THROW".

- **setToHandleRethrowState:** Sets this "ExceptionState"-instance to state "HANDLERETHROW".
- **setToRethrowingState:** Sets this "ExceptionState"-instance to state "RETHROW".
- **rethrow:** Rethrows the currently handled Exception (implicit rethrow).
- **holdsExceptionOfTypeT:** Compares the type of the most recently thrown exception object in the linked list (including the one from this "ExceptionState"-instance) to a given template parameter type. Returns true if the template parameter type is a subtype or equal to the type of the exception object.
- **getExceptionReference:** Returns a pointer to the most recently thrown exception object. It also sets the state of this "ExceptionState"-instance to "HANDLETHROW" or "HANDLERETHROW" respectively, as this method gets called whenever a matching "catch"-block gets entered.
- **throwException:** Gets used to throw a new exception or to explicitly rethrow. There is a copy- and a move-implementation which get used according to the exception-object's constructor definitions. In case a move-constructor is defined, the exception object to throw gets passed as r-value reference to the method to minimize the costs of copying the exception object into the "exceptionBuffer" of this "ExceptionState"-instance.

3.2 Semantic code replacements

The classic exception handling mechanisms with its keywords "try", "catch" and "throw" get implicitly replaced with other inline code segments. This is realized recursively by the Slim-Exception-Plugin, which alters the Abstract syntax tree (AST) of the program at compile time. The following section shows the effective Code replacements for each case.

3.2.1 Try-catch-blocks

At the beginning of each new try-catch section a new ExceptionState-instance is created on the stack and attaches itself to the end of the linked list of the current ExceptionState-instance (Line 2, Listing 3). Goto labels are created for the start and end of the try section. These Labels have a unique postfix ("N", Line 8 & 27) for each try-catch block, to allow for multiple (possibly nested) try-catch blocks in the same function or method. If no exception was thrown by the end of the try block, a goto to the end of the catch-section is executed (Line 7).
Each catch block gets turned into an if (or else if) to check if the type of the exception matches the catch-type (Lines 10 & 16). The catch-all block is represented as final else-block without condition (Line 22). If the try-catch block only has the catch-all block, no check is inserted.
The catch-by-reference catch (Line 12) uses a reference to the exception object which is stored in the exception-buffer of the current ExceptionState-instance, while the catch-by-value catch (Line 18) uses the Copy-Constructor to create a new instance of the exception Object.

```
1                 ->   ExceptionState* __pCurFunctionExcState =
                       ExceptionState::getCurrentExceptionState();
2  try            ->   ExceptionState __esStore(__pCurFunctionExcState);
3  {              ->   {
4  //tryContent   ->   //tryContent (on throw: goto __startCatchLabelN;)
5  }              ->   }
6
7                 ->   goto __endCatchLabelN;
8                 ->   __startCatchLabelN:
9
10 catch(T& e)    ->   if(__esStore.holdsExceptionOfTypeT<T>())
11 {              ->   {
12                ->      T& e = __esStore.getExceptionReference<T>();
```

```
13 //catchContent   ->      //catchContent
14 }                ->   }
15
16 catch(T e)       ->   else if(__esStore.holdsExceptionOfTypeT<T>())
17 {                ->   {
18                  ->      T e(__esStore.getExceptionReference<T>());
19 //catchContent   ->      //catchContent
20 }                ->   }
21
22 catch(...)       ->   else
23 {                ->   {
24 //catchContent   ->      //catchContent
25 }                ->   }
26
27                  ->   __endCatchLabelN:
```

Listing 3: Code replacement for a try-catch-block with different kinds of catches

3.2.2 Throw expression

The exception object to throw gets moved into the currently valid ExceptionState-instance, which either is the "__esStore" (Listing 4, Listing 5 & Listing 6), or the one retrieved by calling "ExceptionState::getCurrentExceptionState" (Listing 7).
After the exception was thrown the execution of the function gets stopped by directly jumping to the handling context, which is either the matching label of the catch section (Listing 4 & Listing 6), or the calling context (Listing 5 & Listing 7), where it is handled as shown in subsubsection 3.2.4.

```
1 T e();        ->   T e();
2 throw e;      ->   __esStore.throwException<T>(std::move(e));
3               ->   goto __startCatchLabelN;
```

Listing 4: Throwing an exception directly in a try-block

```
1 T e();        ->   T e();
2 throw e;      ->   __esStore.throwException<T>(std::move(e));
3               ->   return;
```

Listing 5: Throwing directly in a catch-block which is not surrounded by an outer try-catch-block

```
1 T e();        ->   T e();
2 throw e;      ->   __esStore.throwException<T>(std::move(e));
3               ->   goto __startCatchLabelM; //label from outer try-catch-block
```

Listing 6: Throwing directly in a catch-block which is surrounded by an outer try-catch-block

```
1               ->   ExceptionState* __pCurFunctionExcState = ExceptionState::getCurrentExceptionState();
2 T e();        ->   T e();
3 throw e;      ->   __pCurFunctionExcState->throwException<T>(std::move(e));
4               ->   return;
```

Listing 7: Throwing an exception in a function or method

3.2.3 Implicit rethrow expression

An implicit rethrow is implemented similarly to a direct throw (subsubsection 3.2.2), but the "rethrow" method is called instead. (An explicit rethrow is identical to a direct throw).

```
1
2 throw;     ->    __esStore.rethrow();
3            ->    return;
```

Listing 8: Rethrowing directly in a catch-block which is not surrounded by an outer try-catch-block

```
1
2 throw;     ->    __esStore.rethrow();
3            ->    goto __startCatchLabelM; //label from outer try-catch-block
```

Listing 9: Rethrowing directly in a catch-block which is surrounded by an outer try-catch-block

```
1            ->    ExceptionState* __pCurFunctionExcState = ExceptionState::getCurrentExceptionState();
2 throw;     ->    __pCurFunctionExcState->rethrow();
3            ->    return;
```

Listing 10: Rethrowing an exception in a function or method

3.2.4 Calling potentially throwing functions or methods

After each call to a potentially throwing function or method (not marked as "noexcept") a check is inserted. If an exception was thrown, a direct jump to the handling context is executed, which is either the matching label of the catch section (Listing 11 & Listing 13), or the calling context (Listing 12 & Listing 14). If the function or method has a return value, it gets temporarily stored in a new variable ("__SLIM_EXC_pTmp") and is only assigned to the actual target variable if no exception was thrown.

```
1 x = foo();    ->    auto __SLIM_EXC_pTmp = foo();
2               ->    if(__esStore.isExceptionThrowing())
3               ->       goto __startCatchLabelN;
4               ->    x = __SLIM_EXC_pTmp;
```

Listing 11: Calling potentially throwing functions or methods directly in a try-block

```
1 x = foo();    ->    auto __SLIM_EXC_pTmp = foo();
2               ->    if(__esStore.isExceptionThrowing())
3               ->       return;
4               ->    x = __SLIM_EXC_pTmp;
```

Listing 12: Calling potentially throwing functions or methods directly in a catch-block which is not surrounded by another try-catch-block

```
1 x = foo();    ->    auto __SLIM_EXC_pTmp = foo();
2               ->    if(__esStore.isExceptionThrowing())
3               ->       goto __startCatchLabelM; //label from outer try-catch-block
4               ->    x = __SLIM_EXC_pTmp;
```

Listing 13: Calling potentially throwing functions or methods directly in a catch-block which is surrounded by another try-catch-block

```
1               ->    ExceptionState* __pCurFunctionExcState = ExceptionState::getCurrentExceptionState()
2 x = foo();    ->    auto __SLIM_EXC_pTmp = foo();
3               ->    if(__pCurFunctionExcState->isExceptionThrowing())
4               ->       return;
5               ->    x = __SLIM_EXC_pTmp;
```

Listing 14: Calling potentially throwing functions or methods in another function or method

3.2.5 Additional notes

Return-statements: Everytime the Slim-Exception-Plugin needs to insert a "return" statement to exit the function or method as part of the code replacements, a "return;" without value is inserted. This is also true if the function or method has a return type! In code this usually is not allowed but it is possible as direct modification of the AST. As this only occurs when an exception is thrown, the return value is not used and thus the code is still safe and without side effects.

Call to "getCurrentExceptionState": In every function or method where a reference to the currently valid ExceptionState-instance is needed, it gets only retrieved once by calling "ExceptionState::getCurrentExceptionState" at the beginning of the function or method.

3.3 Slim-Exception-Plugin

This chapter mainly addresses developers who wants to extend or fix bugs in the "SlimExc"-Plugin. It shows how to get a proper toolchain for development, as well as an overview of the internal structure of the plugin.

3.3.1 Development toolchain

In order compile and debug the plugin, the sources of the GCC are needed. They can be downloaded from the project site [6]. To get the related binaries, the sources have to be compiled. To do that, the systems version of the GCC compiler must be installed. In Ubuntu this can be done with the following command:

```
1 sudo apt-get install build-essential
```

Next the binaries must be built. The needed commands can be found in the related documentation [4]. The configure-command

```
1 ./configure --prefix=$HOME/GCC-10.3.0-release --disable-multilib --disable-libmpx
    --enable-languages=c,c++
```

works well in Ubuntu with the GCC Version 10.3.0. With this command path "$HOME/GCC-10.3.0-release/bin" is used for the resulting binaries.
Finally an Integrated Development Environment (IDE) is needed. "Eclipse-CDT" is recommended here. There is a tutorial [3] which can be used to set up "Eclipse-CDT" for editing, compiling and debugging GCC plugins.

3.3.2 Used Plugin Callbacks

The "GCC plugin system" offers a lot of possibilities to alter the compilation process. For that, callbacks can be registered by the plugin. The GCC calls the callbacks on the according compile-stages with some useful parameters. A list of the available callbacks can be found at the GCC plugin documentation site [5]. The Slim-Exception-Plugin currently uses the callbacks listed in Table 1.

Callback	Usage
PLUGIN_INFO	Registering an info-structure with the plugins version and help text
PLUGIN_START_UNIT	Does some checks and set the plugin dependent define directives
PLUGIN_PRE_GENERICIZE	Does the code replacements listed in subsection 3.2
PLUGIN_FINISH_TYPE	Checking the attributes of the Slim-Exception-Library

Table 1: List of used callbacks

3.3.3 Implementation of the AST modifications

As already mentioned in subsubsection 3.3.2 the AST modifications are done in the Callback "PLUGIN_PRE_GENERICIZE". This Callback offers a Tree-Node which refers to the AST of the currently parsed function or method. The class "TreeParser" with the method "parse_tree" (in "TreeParser.hpp") is used to traverse this AST. The "parse_tree"-method can receive a callback-Function as parameter, which is called for every subnode of the AST. It is used to print the AST as well as to modify it. The modifications are done in the class "ConvertTools". Its method "modifyAST" is used as callback for the "parse_tree"-method and does the modification of the AST depending on the type of the currently traversed node. In some cases with nested exception code structures (try, catch, ...), new instances of the "ConvertTools"-class are created to traverse sub-trees of the AST. They can access their parent instances through a linked list, which allows to retrieve relevant information about the context in some cases.

3.3.4 Compiletime errors and messages

In order to prevent faulty code and undefined behavior the Plugin analyses the AST for possible Errors and outputs matching compiletime messages:

Message	Description
Multiple inheritance is not supported for exceptions!	When a multiple inheriting class-instance gets thrown.
ExceptionState-Class is not available here. Please insert the needed include (SlimExcLib.hpp) or exclude the code with Filters	When an exception handling mechanism is used without including the SlimExc-Library.
Current configuration only allows fundamental types for exception handling! The type X is not allowed!	When a non fundamental type is used while this functionality is disabled in the configuration (see subsection 4.2).
Current configuration only allows type X for exception handling! The type Y is not allowed!	When a wrong type is used while the configuration only allows a specific fundamental type as exception (see subsection 4.2).
The current configuration allows only to throw one Type! Please only catch this one!	When multiple catch handlers exist while the configuration only allows one specific fundamental type as exception (see subsection 4.2).
The type X does not fit in the exceptionBuffer! Maybe you need to extend the Buffer by Plugin-configuration.	When a type is thrown which is larger than the Exception-Buffer which can be configured in the configuration (see subsection 4.2).
An exception may leave a "noexcept"-section, which always results in std:terminate!	When an exception could be thrown inside a "noexcept"-function. This can either happen by directly throwing or rethrowing inside a "noexcept"-function, or by calling a function which is not marked "noexcept". This includes Constructors and Destructors as well. However, the C++-Standard allows it.*
Possibly uncaught Exceptions	When a try-catch-block doesn't include a catch-all-handler.*

* The level of this messages can be configured in the configuration (subsubsection 4.2.1).

3.4 SlimRTTI-System

In order to catch exceptions polymorphycally a Runtime Type Information (RTTI)-System is required. Though it is possible to use the standard RTTI-Implementation, that might not be desirable because it comes with a significant memory overhead for each object. For this reason the Slim-Exception-System offers an alternative RTTI-Implementation which only has a minimal memory overhead for each type which gets thrown. This RTTI-Implementation is only able to compare types at runtime, but does not support dynamic casting and reflection.

It also does not support multiple and virtual inheritance.

3.4.1 Header overview

The entire SlimRTTI-System is implemented in the file "SlimRTTI.hpp", inside the namespace "SlimRTTI". An overview is given in Listing 3.4.1.

```
1  namespace SlimRTTI
2  {
3     template<typename T> inline void** getTypeId() noexcept;
4     template <typename T> inline constexpr uint8_t getPointerLevel() noexcept;
5     template <class T> static inline constexpr bool isInherited() noexcept;
6
7     class InstanceType
8     {
9     private:
10       void** typeId = nullptr;
11       union MetaData
12       {
13          uint16_t data;
14          struct Fields
15          {
16             uint8_t constMask;
17             uint8_t ptrDepth;
18          } fields;
19       } metaData = {};
20
21    public:
22       InstanceType() noexcept;
23       void inline clear() noexcept;
24       InstanceType& operator=(const InstanceType& other) noexcept;
25       template <class T> void set() noexcept;
26       template <class T> inline bool isEqualTo() noexcept;
27       template <class T> inline bool isDerivedOf() noexcept;
28       template <class T> inline bool isBaseOf() noexcept;
29       template <class T> bool do_catch() noexcept;
30 }
```

Functions in Namespace "SlimRTTI":

- **getTypeId:** Returns a unique ID for the plain type of the template-specified type. All Qualifiers ("&", "*", "const" and "volatile") are dismissed. See subsubsection 3.4.2 for more details on the implementation.
- **getPointerLevel:** Return the pointer level of the template-specified type. References are not counted as pointers. Types which are no pointer have a pointer level of 0.
- **isInherited:** Returns "true" if the template specified type has a base class.

Classes in Namespace "SlimRTTI":

- **class InstanceType:** This class represents the type of an instance at runtime. It includes the ID of the plain type as well as it's pointer- and const-qualifiers.

  Attributes in "InstanceType":

  - **typeId:** The ID of the plain type.
  - **metaData:** Contains additional information about the pointer- and const-qualifiers.

* **constMask:** A bitmask which represents the constness for each pointer level and the constness for the plain type. A "1" represents a const qualifier, a "0" it's absence. This bitmask is the reason the maximal supported pointer level is 7.
* **ptrDepth:** This value represents the pointer level (the number of "*") of the type.

Methods in "InstanceType"

- **InstanceType:** Constructor. Sets the instance to be empty, representing type "void".
- **clear:** Sets the instance to be empty, representing type "void".
- **operator=:** Assignment operator.
- **set:** Sets the instance to represent the template-specified type, including it's pointer- and const-qualifiers.
- **isEqualTo:** Compares the internally represented plain type to the template specified type. All qualifiers ("&", "*", "const" and "volatile") are disregarded for this comparison. It returns true if the two plain types are identical, no one being the base of the other.
- **isDerivedOf:** Checks if the internally represented plain type is derived from the template specified type. All qualifiers ("&", "*", "const" and "volatile") are disregarded for this check. It returns true only if the internally represented plain type is derived from the template specified type, false if it is the other way around or if the two types are identical.
- **isBaseOf:** Checks if the internally represented plain type is a base of the template specified type. All qualifiers ("&", "*", "const" and "volatile") are disregarded for this check. It returns true only if the internally represented plain type is a base of the template specified type, false if it is the other way around or if the two types are identical.
- **do_catch:** Checks if the internally represented plain type can be caught by a catch-clause with the template specified type. Pointer- and const-qualifiers are taken into account.

3.4.2 Generation of unique type-IDs

The core idea to generate unique IDs for each type is to use the address of a variable as number. This guarantees the value to be system-wide unique and constant. It also allows the actual value of the variable to be used otherwise. By declaring the variable as a pointer, it can point to the ID of it's basetype, which enables non-multiple inheritance. If the type has no basetype, the pointer is just a nullptr. All IDs of a type hierarchy thus form a linked list, which can be traversed for polymorphic type comparisons.
This mechanism is implemented in "getTypeId", which is shown in Listing 3.4.2. As this template function gets instantiated for each type "T", a unique static const variable "id" is generated for each value of "T". The linked list of IDs for polymorphic types gets generated by calling itself recursively. Most of the terms in this function are evaluated at compiletime, leaving only the return statements to be executed at runtime.

```
1  template<typename T> inline void** getTypeId() noexcept
2  {
3     //... remove all Qualifiers and Pointers from T...
4
5     if constexpr (std::tr2::direct_bases<T>::type::empty::value)
6     {
7        static void** const id = nullptr;
8        return reinterpret_cast<void**>(const_cast<void***>(&id));
9     }
10    else
11    {
12       typedef typename std::tr2::direct_bases<T>::type::first::type BaseType0;
13       static void** const id = getTypeId<BaseType0>();
```

```
14        return reinterpret_cast<void**>(const_cast<void***>(&id));
15    }
16 }
```

4. Usage

4.1 Environment

The Slim-Exception-System consists of the compiled GCC plugin (binary "SlimExc"), the Slim-exception handling library (SlimExcLib) and 3 user defined functions (subsubsection 4.1.4).

4.1.1 GCC and Platform compatibility

The current version of the Slim-Exception-Plugin (0.9.0) is successfully tested with the following versions of the GCC on the following platforms:

Development platform / Target platform	x64 Linux*	x64 Windows
x64 Linux*	10.1.0, 10.2.0, 10.3.0, 10.4.0, 11.1.0, 11.2.0, 11.3.0	-
x64 Windows	-	-
ARM-none-eabi	10.3.1	-

* Tested on Ubuntu18.04 and OpenSUSE Tumbleweed

The same plugin-binary was used on all platforms. It is not unlikely the Slim-Exception-Plugin will also work on other target platforms. However, it is confirmed that GCC-Versions 9.* and 12.* do not work properly to full extent. For testing it with unverified GCC-versions the version check can be disabled as shown in Listing 19.

4.1.2 Toolchain

The g++ call scheme to compile your source-File with the Slim-Exception-Plugin is shown in Listing 15.

- The parameter "-fplugin" specifies the Path to the Slim-Exception-Plugin Executable and enables its usage.
- The plugin parameters are listed in subsubsection 4.2.1. If not specified, default values will be used.
- C++-17 or newer must be used.
- The optional GCC parameter "-fno-rtti" is used to disable the C++ runtime type information system. It may only be used if the Slim-Exception-Plugin is configured to use its own SlimRTTI mechanism instead (subsection 4.2).
- In order for the Slim-Exception-System to work properly it is necessary to keep the C++ standard exception handling enabled. The flag "-fno-exceptions" must not be used.
- All other GCC-Flags and GCC-Parameters as well as the linker can be used as usual.

```
1 g++ -fplugin={pathToPlugin}/SlimExc [plugin parameters] -std=c++17 [-fno-rtti] [other gcc parameters]
      {pathToCppFile}
```

Listing 15: Call scheme of the C++-Compiler with the Slim-Exception-Plugin

4.1.3 Library

The Slim-Exception handling library needs to be included in all project files which use exception handling mechanisms. It may be convenient to include the library in form of a git submodule (Listing 16).

```
git submodule add https://oko.spdns.de/git/admin/projects/oko/DeterministicExceptionHandlingLibrary
```

Listing 16: Command to add the library as git submodule to your project

4.1.4 User defined functions

In order for the system to function, the user is required to provide an implementation of the functions "ExceptionState::getCurrentExceptionState", "ExceptionState::setCurrentExceptionState" in the namespace "SlimExcLib" and "std::terminate" if not already defined.
The implementation must provide an initial ExceptionState object as well as an updatable pointer which initially points to the initial ExceptionState object for each thread. A minimal example implementation for a single threaded system is shown in Listing 17.

- "SlimExcLib::ExceptionState::getCurrentExceptionState" must return a pointer to the active ExceptionState object of the current thread.
- "SlimExcLib::ExceptionState::setCurrentExceptionState" must update the pointer to the active ExceptionState object of the current thread.
- "std::terminate" see https://en.cppreference.com/w/cpp/error/terminate

```
#include "SlimExcLib.hpp"

namespace SlimExcLib
{
   ExceptionState baseExceptionState(NULL);
   ExceptionState* pCurrentExceptionState = &baseExceptionState;

   ExceptionState* ExceptionState::getCurrentExceptionState(void) noexcept
   {
      return pCurrentExceptionState;
   }

   void ExceptionState::setCurrentExceptionState(ExceptionState* newInstance) noexcept
   {
      pCurrentExceptionState = newInstance;
   }
}

[[noreturn]] void std::terminate() noexcept
{
   while(1){};
}
```

Listing 17: Minimal example implementation of the necessary user defined functions

4.1.5 Other dependencies

The Slim-Exception-System depends on a definition of the placement-new and placement-delete operator. Those are declared as "extern" in the Slim-Exception-Library. Other than that there are no dependencies to the C++-standard-library and the code can be compiled with the "-nodefaultlibs"-flag.

4.2 Configuration

It is possible to change the behavior of the Slim-Exception-System by using either commandline parameters or a XML-Configuration File. The default configuration is set to offer a maximal standard-compatibility as described in subsection 4.3. It is possible to restrict the features of the System for a possibly better performance and code size.

4.2.1 Parameters

XML	CMD-Parameter	Values/Description
	*-config=	Path to XML-Configuration-File
Exceptions/ SlimExceptions	*-f-SlimExc *-fno-SlimExc	**Enabled**/**Disabled** Enables/Disables the usage of the Slim-Exceptions (The Plugin itself still remains enabled but the default Exception-Code is not modified)
Exceptions/ ThrowableTypes	*-ThrowableTypes=	**All** - Maximal Functionality, allows throwing all Instances (most memory and runtime costs) **Fundamental** - Restricts the system to only allow throwing non-class-Types (reduced memory and runtime costs) **Single** - Restricts the system to only allow throwing a single Type, which is specified by "SingleType". (Minimal memory and runtime costs)
Exceptions/ SingleType	*-SingleType=	**char, schar, uchar, ushort, short, uint, int, ulong, long, ulonglong, longlong, float, double** Defines the type which is allowed to be thrown if "ThrowableTypes"="Single".
Exceptions/ Buffersize	*-Buffersize=	Range: **1 - n**, Default: **10** The size of the Exceptionbuffer in Bytes.
RTTI/ SlimRTTI	*-f-SlimRTTI *-fno-SlimRTTI	**Enabled**/**Disabled** Enables/Disables the usage of the SlimRTTI-System for the Exception-matching. If Disabled, the default RTTI-System must be enabled instead (not using -fno-rtti).
RTTI/ PointerTypes	*-f-PointerTypes *-fno-PointerTypes	**Enabled**/**Disabled** Enables/Disables Pointer-Types in the SlimRTTI-System. (Reduced memory and runtime costs if Disabled)
RTTI/ Qualifiers	*-f-Qualifiers *-fno-Qualifiers	**Enabled**/**Disabled** Enables/Disables Qualifiers (const) in the SlimRTTI-System. If Disabled, const-Qualifiers are not taken into account on Exception-matching! (Reduced memory and runtime costs if Disabled)

Filters/ InitIgnoreFile	*-f-InitIgnoreFile *-fno-InitIgnoreFile	**Enabled**/**Disabled** Enables/Disables the initialization of the IgnoreFile with a list of used Namespaces, Classes and Functions/Methods. If Enabled, the Slim-Exception-System is automatically disabled. This can be used as a starting point for a custom Blacklist/Whitelist to exclude partial code from being handled with the Slim-Exception-System.
Filters/ InitFiltered	*-f-InitFiltered *-fno-InitFiltered	**Enabled**/**Disabled** Enables/Disables filtering for the initialization of the IgnoreFile. If Enabled, the current Blacklist/Whitelist is already applied to initialization.
Filters/ WhitelistStrategy	*-f-WhitelistStrategy *-fno-WhitelistStrategy	**Enabled**/**Disabled** Enabled = Whitelist Disabled = Blacklist
Filters/ IgnoreFile	*-IgnoreFile=	Default: **empty** Path to the XML-Filter-File which is used as Blacklist/Whitelist to exclude partial code from being handled with the Slim-Exception-System. This path is also used for initializing the File when "InitIgnoreFile" is enabled. If empty, filtering is disabled completely.
Diagnostics/ WrongGCCVersion	*-WrongGCCVersion=	**Ignore, Info, Warning, Error** Message when the GCC-Version is not compatible with the plugin binary. It might still work, so it is possible to decrease the Diagnostic-Level. But this can lead to undocumented problems!
Diagnostics/ StdTerminateOn Exception	*-StdTerminateOn Exception=	**Ignore, Info, Warning, Error** Message when an exception is thrown which is never caught, leading to a guaranteed call of "std::terminate".
Diagnostics/ PossiblyUncaught Exception	*-PossiblyUncaught Exception=	**Ignore, Info, Warning, Error** Message when an exception could propagate out of a non-throwing region. This can be the case when a "try-catch"-block within a "noexcept" section doesn't include a catch-all handler.
Debugging/ Verbose	*-f-Verbose *-fno-Verbose	**Enabled**/**Disabled** Enables/Disables additional outputs while traversing the AST.
Debugging/ GenerateAST	*-f-GenerateAST *-fno-GenerateAST	**Enabled**/**Disabled** Enables/Disables the printing of the AST. Two files in the Debugging-Directory are generated for each function, one with the unmodified and one with the modified AST.
Debugging/ Directory	*-Debugging-Directory=	Default: **empty** Path to the Debugging-Directory, is used by "GenerateAST".

* Commandline Parameter Prefix: -fplugin-arg-SlimExc
_ underlined: Default Values.

4.2.2 Examples

The following examples show the usage of some of the command line parameters:

```
g++ -fplugin={pathToPlugin}/SlimExc -fplugin-arg-SlimExc-config={pathToPluginConfig}/pluginConfig.xml
    -std=c++17 -O3 -g3 -Wall -fno-rtti -c -o "myFile.o" myFile.cpp
```

Listing 18: Using the XML-Config-File

```
g++ -fplugin={pathToPlugin}/SlimExc -fplugin-arg-SlimExc-WrongGCCVersion=Warning -std=c++17 -c -o
    "myFile.o" myFile.cpp
```

Listing 19: Using a different GCC-Version (be aware, this can result in undocumented error-Messages)

```
g++ -fplugin={pathToPlugin}/SlimExc -fplugin-arg-SlimExc-IgnoreFile={pathToMyFilterFile}/ignore.xml
    -fplugin-arg-SlimExc-f-InitIgnoreFile -std=c++17 -c -o "myFile.o" myFile.cpp
```

Listing 20: Generating found Entries in a Filter-File

```
g++ -fplugin={pathToPlugin}/SlimExc -fplugin-arg-SlimExc-IgnoreFile={pathToMyFilterFile}/ignore.xml
    -fplugin-arg-SlimExc-f-WhitelistStrategy -std=c++17 -c -o "myFile.o" myFile.cpp
```

Listing 21: Using a Filter-File as Whitelist

```
g++ -fplugin={pathToPlugin}/SlimExc -fplugin-arg-SlimExc-ThrowableTypes=Single
    -fplugin-arg-SlimExc-SingleType=short -std=c++17 -c -o "myFile.o" myFile.cpp
```

Listing 22: Allow only throwing "short"-Types

```
g++ -fplugin={pathToPlugin}/SlimExc -fplugin-arg-SlimExc-Debugging-Directory={pathToDirectory}
    -fplugin-arg-SlimExc-f-GenerateAST -std=c++17 -c -o "myFile.o" myFile.cpp
```

Listing 23: Print the ASTs for Debugging in a Directory

4.2.3 Using filtering to exclude Code sections

In order to exclude precompiled libraries or legacy code without "noexcept"-declarations it is possible to use an ignorefile (as shown in Listing 21).
To avoid having to write the ignorefile manually, it is possible to use the "initIgnoreFile"-Option as shown in Listing 20. This initializes the ignorefile's XML-Structure and generates a list of all Namespaces, Classes and Functions which can be copy-pasted into the blacklist/whitelist section of the ignorefile.
Adding an empty XML-Tag for a Namespace or a Class selects everything inside of this Namespace or Class as well.

4.3 Deviations from C++-Standard

The following deviations are based on the assumption that Slim-Exception-Plugin is configured with its default values which represent the full function range. It is possible to reduce the function range and thus the C++ standard conformity in favor of better performance in the configuration (subsection 4.2).

4.3.1 Working with precompiled code

If standard exception are thrown from precompiled code (without the Slim-Exception-Plugin), those can not be handled by the Slim-Exception-System! Equally no exceptions from the Slim-Exception-System can be handled by precompiled code! Further it is highly recommended to exclude all precompiled code by the "ignoreList" configuration in order to avoid unnecessary runtime overhead (subsubsection 4.2.1).

4.3.2 Multiple and virtual inheritance

Multiple and virtual inheritance is currently not supported for classes which get thrown as exception objects by the system.

4.3.3 Noexcept keyword

To minimize the runtime overhead it is highly recommended to use the official C++ keyword "noexcept" on all function and method definitions which do not throw. Unlike the C++ standard behavior, the Slim-Exception-System does not allow exceptions to propagate through functions or methods which are marked as "noexcept".

4.3.4 "Extern C" is treated as "noexcept"

All functions declared as "extern C" are treated as if declared "noexcept(true)". Throwing from "extern C"-Functions is not allowed with the Slim-Exception-System.

4.3.5 Stack unwinding in case of std::terminate

When a condition is met which leads to a call of "std::terminate", the call is immediately executed without a preceding stack unwinding. Thus, the objects on the stack don't get properly destructed. In practice this should not be a problem, because a call to "std::terminate" ends the execution of the program. However, if the Destructors are used to disable some physical periphery this can be a critical deviation from the standard. It is recommended to implement all necessary hardware shutdowns in the "std::terminate"-routine.
In [1, page 8] it is argued that the current c++-standard for exception handling cannot be implemented in a space- and time-deterministic manner, because there is no upper bound to the number of active exceptions at any given point in time. By introducing the above deviation from the standard, the upper bound for the number of active exceptions is set to be equal to the number of nested try-Blocks in the system.

4.3.6 Exception Size limit

The size of all thrown exception object must be lower or equal to the configured Buffer Size (buffersize in configuration subsubsection 4.2.1).

4.3.7 Maximal pointer depth

If the SlimRTTI-System is used, the maximal pointer depth of thrown exceptions is 7.

4.4 Known issues

- On arm-none-eabi systems, an Implementation of the functions "__aeabi_unwind_cpp_pr0" and "__aeabi_unwind_cpp_pr1" is required but may be left empty.

4.5 Bug reporting

If any bugs are encountered, please follow these instructions to file a report:

1. Compile the program without the Slim-Exception-System and verify it's correctness to make sure the error actually is in the Slim-Exception-System.
2. Extract a minimal example which reproduces the bug from your code.
3. Send an E-Mail including
   - the minimal code example
   - a description of the bug
   - the version numbers of SlimExc and GCC
   - the target- and build platforms
   - any possibly occurring compiler- or runtime errors

   to developer@philipp-rimmele.de

5. Benchmarks

All benchmarks were recorded on a "STM32F4"-Microcontroller (arm-none-eabi) with the default configuration (subsection 4.2). As of now, only the code size and runtime of individual statements were compared. A more in-depth investigation of the effective performance on larger examples as well as ram memory usage is still to be conducted. All measured values are samples from a minimal example.

5.1 Code size

Comparison between the code size of different exception statements:

	Optimization O0		Optimization O3	
	Default	**SlimExc**	**Default**	**SlimExc**
Exception Library	12391	156	12340	84
Try-catch-all	8	28	4	20
Try-catch→throw-instance	36	80	36	88
Throw int	0	82	36	60
Rethrow int	0	12	8	44
Call of throwing function	4	20	4	12

All values in bytes.

The benchmarks show that the default implementation has a more significant code size overhead upon it's initial usage due to it's library size, while the SlimExc implementation has a larger code size overhead for each exception statement. The overall code size will thus heavily depend on the project size and the number of exception statements.
The SlimExc-System allows to compile the project without default-libraries, which account for the major library size overhead on the default exception system.

5.2 Execution speed

Comparison between the runtime of different exception statements:

	Optimization O0		Optimization O3	
	Default	**SlimExc**	**Default**	**SlimExc**
Entry in try-block	3	151	0	37
Exit from try-block	4	176	0	62
Full try-block	4	300	0	97
Function call	26	61	16	34
Throw int →catch all	4904	437	5100	57
Throw int →catch int	6044	569	5587	86
Throw instance →catch instance	6796	635	6305	90
Throw instance from function →catch instance	9768	708	7960	123
Throw instance →propagate up →catch all	5637	1391	7125	208
Rethrow int implicite →catch all	6801	883	6530	144

All values are processor-ticks on a "STM32F4"-Microcontroller.

The benchmarks show that the default implementation has a major runtime overhead on the error path and nearly no overhead on the success-path, while the SlimExc implementation has a more evenly distributed runtime overhead.

6. The future of SlimExc

The current version of SlimExc was developed without any funding just as hobby- and research project. Though it is fully functional, there are many possibilities for future optimizations and improvements which we would like to develop if any funding opportunities occur. The following list of possible ideas is non-exhaustive:

- **Quality assurance:** Restructure and extend tests and set up continuous integration.
- **Quality assurance:** Continuous bug fixing and maintenance.
- **Compatibility:** Support other development- and target platforms.
- **Compatibility:** Support other GCC-Versions.
- **Optimization:** Compiler Warning/Error for functions which are not throwing but also not marked as "noexcept".
- **Optimization:** Many performance and code size optimizations for various cases.
- **Configuration Option:** Define "noexcept(true)" to be the default-value for all functions.
- **Configuration Option:** Offer an alternative implementation of the "ExceptionState"-class which allows the usage of dynamic memory allocation for the exception object (for example a deterministic memory pool).
- **Configuration Option:** Make maximally supported pointer depth configurable with SlimRTTI.
- **Feature:** Support multiple inheritance for exception objects.

- **Feature:** Automatic generation of wrapper functions for precompiled code with default exceptions for better compatibility with external libraries.
- **Feature:** Extend SlimRTTI for more use cases outside of exception handling.

Acronyms

AST Abstract syntax tree. 9, 12, 13, 19, 24

GCC GNU Compiler Collection. 4, 12, 15, 16, 24

IDE Integrated Development Environment. 12

RTTI Runtime Type Information. 13

Bibliography

[1] Herb Sutter. *Zero-overhead deterministic exceptions: Throwing values*. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0709r0.pdf. 2018.

[2] Nico Brailovsky. *C++ Exception Handling Internals*. https://monoinfinito.wordpress.com/series/exception-handling-in-c/.

[3] Stephan Friedl. *Tutorial to set up Eclipse for GCC plugin debugging*. https://stephanfr.com/2013/05/19/building-gcc-plugins-part-1-c-11-generalized-attributes/.

[4] *GCC installing documentation*. https://gcc.gnu.org/wiki/InstallingGCC.

[5] *GCC plugin documentation*. https://gcc.gnu.org/onlinedocs/gcc-4.5.0/gccint/Plugins.html.

[6] *GCC Project site*. https://gcc.gnu.org/.

[7] Björn Frankec James Renwick Tom Spink. "Low-Cost Deterministic C++ Exceptions for Embedded Systems". In: *Proceedings of the 28thInternational Conference on Compiler Construction (CC '19), February 16–17,2019, Washington, DC, USA.ACM, New York, USA* (2019). https://www.research.ed.ac.uk/portal/files/78829292/low_cost_deterministic_C_exceptions_for_embedded_systems.pdf.

[8] Daniel Marjamäki. https://github.com/danmar/gcc-plugins/blob/master/dump-tree-xml/dump-tree-xml.c.

[9] Herb Sutter. https://herbsutter.com/2016/09/25/to-store-a-destructor/. 2016.

Credits and licenses

Acknowledgments to

- Herb Sutter for his theoretical analysis of the C++ exception system [1] and his cool trick to get destructor pointers [9].
- James Renwick, Tom Spink and Björn Franke for their proof of concept [7].
- Stephan Friedl for his helpful GCC plugin debugging tutorial [3].
- Daniel Marjamäki for his reference implementation of the GCC AST traversion [8].
- Yves Berquin for the TinyXPath-Library which is used by the SlimExc-Plugin.