UNIVERSITY OF TWENTE.

Academic Research Skills
Conference Proceedings

internal edition

Adding a feature to the C language – Exception Handling

Catalin Chiprian, Andi-Stefan Serpe

Supervisor: Dr. Peter Lammich

Pre-master track: CS and IST

Submission date: January 24, 2025

This paper explores the integration of exception handling into the C programming language, addressing its inherent limitations in error management. Traditional approaches, such as return codes and the errno variable, lack contextual richness, require manual intervention, and complicate code clarity. Unlike modern languages like C++ and Java, which utilize structured exception handling with features such as try-catch blocks and automatic resource cleanup, C lacks a standardized mechanism for efficient error handling.
To address this gap, we propose modifying the widcc compiler to natively support exception handling. The implementation introduces new language constructs aligned with modern exception-handling paradigms, supported by detailed documentation of the modifications made to the compiler. The functionality is rigorously validated through comprehensive test cases encompassing diverse exception-handling scenarios in C.
This research demonstrates the feasibility of augmenting C's capabilities, enabling safer and more maintainable error management while paving the way for further advancements in C compiler design.

Exception Handling in C; Widcc Modification; Exception Implementation; Hacking Compiler

1. Introduction

Error management is a critical aspect of software development, particularly regarding the maintainability of the programs. However, the C programming language, despite its foundational role in modern computing, lacks a standardized mechanism for structured exception handling. Developers rely instead on alternative solutions such as return codes, the errno variable [4], [5], or macros based on the setjmp/longjmp mechanism [1], [2], [3]. While functional, these methods present significant limitations, including reduced code clarity and challenges in maintaining larger codebases.

In contrast, programming languages such as C++ and Java have adopted structured exception handling constructs like try, catch and finally. These provide a cleaner approach to managing errors. Despite their effectiveness, C has yet to adapt a similarly standardized exception handling system, leaving developers to rely on less efficient alternatives.

This paper addresses this gap by introducing exception handling into C through modifications to the widcc compiler [7]. Widcc`s lightweight architecture, which generates x86 assembly code without intermediate representations, makes it an ideal foundation for experimentation and the implementation of exception handling.

The primary objectives of this research are modifying the compiler to support exception handling by introducing the new constructs (try, catch, finally) and by extending the parsing AST construction and code generation phases to support these constructs.

This work bridges the gap between C`s legacy and the need of modern programming paradigms, enhancing its relevance in contemporary software development. The results demonstrate the feasibility of enriching C with structured exception handling, while offering insight into compiler design and future improvements.

2. Methodology

A compiler is a program that translates high-level source code written in a programming language into a low-level language, such as assembly, that can be executed by a computer's processor. The process typically involves several phases: lexical analysis, in which the source code is tokenized; syntax analysis, which checks the structure of the code based on language rules; semantic analysis to ensure that the program's logic is valid; and finally, code generation and optimization to produce efficient output. Most modern compilers like Clang [6] use intermediate representations (IR) like LLVM IR, as an intermediate step between high-level code and machine code. This approach enables modularity, reuse of optimization techniques, and portability across different hardware architectures.

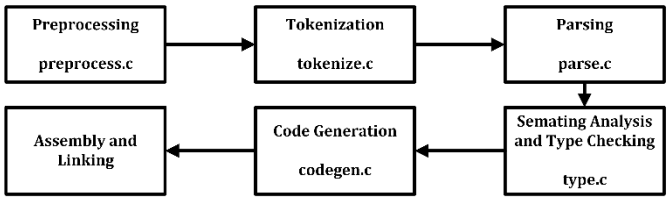

Figure 2.1 Flowchart of the WidCC Compiler Workflow

In contrast, widcc takes a more direct approach (Figure 2.1) by generating x86 assembly code without relying on intermediate representations such as LLVM. This design makes widcc smaller and simpler than compilers like Clang, but also means that widcc

requires custom implementations for tasks that would otherwise use the LLVM backend. For example, in widcc, parsing directly creates an Abstract Syntax Tree (AST), which is then converted into an x86 assembly during the code generation phase. This tightly coupled process eliminates the need for an intermediate representation, reducing complexity but limiting extensibility and cross-platform support.

Until now, exception handling in C has been implemented primarily using macros [1], [2], [3], which lack the syntactic clarity and robustness that structured constructs such as try, catch, and finally provide. For example, exception handling using setjmp and longjmp or custom macros often results in cumbersome, error-prone code that is difficult to maintain. While Microsoft's MSVC [8] compiler introduced structured exception handling (_try, _catch and _finally), this functionality is proprietary and not available as open source. This limitation created a significant gap for researchers and developers who wanted to explore exception handling mechanisms in C. By building on widcc, a lightweight open-source compiler, we have closed this gap and made it possible to implement and explore structured exception handling in an accessible and reproducible manner.

To ensure reproducibility, we documented all phases of the implementation process, including changes to parsing logic, AST construction, and code generation. The phases of the implementation along with the test code can be analysed and accessed at https://github.com/CatalinSaxion/exceptional_widcc. The code demonstrates the syntax and flow of try, catch, and finally, along with detailed explanations of how unique labels manage control flow and how exceptions are stored and loaded based on their types.

3. Results

The parser was extended to support a try-catch-finally exception handling mechanism by introducing new parsing rules for try, catch, and finally blocks. When parsing a try block, it generates a unique label, saves the current Variable Length Array (VLA) state – where a VLA is an array whose size is determined at runtime instead of compile time, allowing for dynamic memory allocation and flexibility in managing array sizes – for a shared scope across the try-catch-finally mechanism. The parser also manages the exception handling state (current_ex_state), a global variable that tracks whether the program is currently IN_TRY_BLOCK or IN_FINALLY_BLOCK. This state is critical for handling logic like return statements and determining whether the user is exiting from a try, catch, or finally block. For example, return statements are linked to the active try block only if they occur within a try or catch, while returns inside a finally block are disallowed, enforcing correct exception handling semantics. The parser then processes the optional catch and finally blocks, ensuring that the catch type matches the thrown exception type. The code supports nested try-catch blocks by maintaining a stack of active try contexts, allowing each nested block to have its own exception state and scope. Additionally, it ensures that exceptions are only thrown within valid try blocks and links throw statements to their corresponding try blocks.

The code generation was extended to support try-catch-finally with the use of unique labels for each try block and generating the corresponding machine instructions (jmp) at compile time targeting the respective labels to manage the control flow. For the try block, the generated code includes a jump to the corresponding finally, if present, otherwise to the end label to ensure proper cleanup when the block concludes without exceptions. In contrast, jumps to the catch block are only generated by the throw statement during exception handling. When encountering a throw statement, code is generated to evaluate the thrown expression and store its value in the appropriate register, determined by the type of the exception variable, followed by a final jump to the corresponding catch block. The catch block supports type-specific handling of exceptions by inspecting the type of the caught variable and generating code to store the exception value. For handling scalar types like integers and floats direct register-to-memory instructions are used, however for composite types like structs and arrays specialized memory copy operations are used. Boolean types are normalized before storage. Nested try-catch blocks are supported by associating each block with a unique label and context, ensuring correct jumps even within nested scopes. Additionally, the finally block ensures cleanup by executing its code regardless of whether an exception is caught. This design ensures proper resource management and variable accessibility, even within nested try-catch-finally blocks.

The return mechanism was modified both in parsing and code generation phases to ensure the execution of a finally block, even when a return is used inside a try or catch. During parsing, if a return is detected within a try or catch block containing a finally, it is linked to the corresponding try node. In code generation, instead of the return instruction (Line 91, Figure 2.3), the return value is stored in the RBX register (Line 76, Figure 2.3), and the control flow jumps to the finally block (Line 77, Figure 2.3). Inside the finally, the cleanup code executes, and the return value is restored from RBX to RAX before exiting (Line 82 - 88, Figure 2.3). In widcc, the register RBX is typically not used by the compiler for general-purpose operations. This design choice allows RBX to be reserved exclusively for the temporary storage of return values when a FINALLY_BLOCK is present. This design ensures proper resource cleanup and consistent behaviour regardless of early exits via return.

The flowchart in Figure 2.2 illustrates the control flow of a program that uses structured exception handling, including try, catch, and finally blocks. Execution begins at the TRY_BLOCK, where semantics are generated for the node as it is entered. If a throw statement is encountered within the TRY_BLOCK, the control flow immediately jumps to the CATCH_BLOCK via a direct jump (jmp) to the catch block's label, where the semantics for the CATCH_BLOCK are generated. If no throw is encountered, the program continues its execution to the FINALLY_BLOCK, generating semantics for it as well.

In the event of a return statement within either the TRY_BLOCK or CATCH_BLOCK, the default behaviour of the return statement is modified. Instead of directly retrieving the return value from the RAX register, the value is stored in the RBX register to allow the FINALLY_BLOCK to execute prior to the return operation. If a FINALLY_BLOCK is present, control flow jumps to the FINALLY_BLOCK, ensuring that the finally semantics are always executed. This redirection guarantees that resource cleanup or other finalization logic specified in the FINALLY_BLOCK is performed, even in the presence of a return statement.

Upon entering the FINALLY_BLOCK, the program generates the corresponding semantics and checks whether the RBX register contains any values. If the RBX register is not empty, indicating a pending return, the value is retrieved from RBX and moved to the RAX register. The program then resumes the default return behaviour, which includes updating the stack frame and returning control to the caller. Conversely, if the RBX register is empty, the FINALLY_BLOCK continues its normal flow.

This design ensures that the semantics of the finally construct are preserved, allowing it to execute regardless of whether a throw or return statement is encountered in preceding blocks. The use of the RBX register as a temporary storage for return values ensures that return behaviour remains consistent, even when a FINALLY_BLOCK is present.

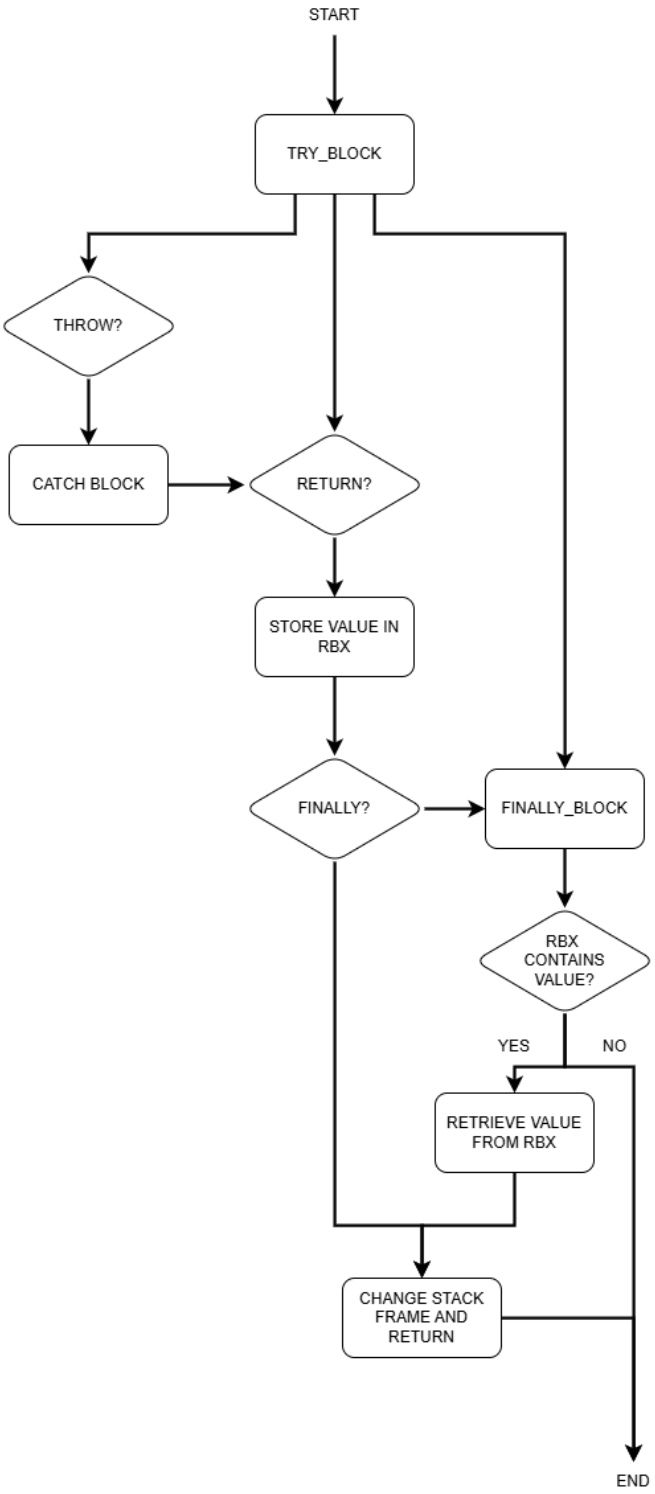

Figure 2.2 Try-Catch-Finally Flowchart

Overall, this [illegible] for using the exception handling mechanism. A throw statement is only valid inside a try block and must include a valid exception expression. While catch blocks are optional, they require exact type matching between the caught and thrown exceptions, ensuring type safety. A finally block, also optional, is provided for cleanup operations, but users are responsible for managing resources appropriately. Returning from a try or catch block executes the finally block, if present, before the function exits; however, return is strictly forbidden within finally. Misuse of these constructs, such as mismatched types or illegal operations, results in clear compile-time errors.

The assembly code snippet in Figure 2.3 illustrates the behaviour of a simple function, test, that contains a try-catch-finally structure with a return statement inside the try block. Comments beginning with the # symbol indicate the corresponding block of code, making it easier to follow the control flow. As described previously and shown in the flowchart in Figure 2.2, encountering a return within the try or catch block modifies its standard behaviour. In this example, the return occurs in the try block.

Initially, the return value is stored in the RAX register as per the default behaviour (line 75). However, the modified behaviour is evident as the value in RAX is also copied into the RBX register (line 76). This additional step ensures the finally block is executed before completing the function. Control then jumps directly to the finally block's label (.L..0finally) at line 77, bypassing the catch block entirely. While the catch block is defined with generated semantics (line 79), it is skipped in this instance because no throw occurs.

Within the finally block (lines 82–88), the program checks whether the RBX register is empty (line 83). If RBX is empty, indicating no pending return, the flow jumps to the end of the finally block (.L..0end) and proceeds to the subsequent code. If RBX is not empty, meaning a return value is pending, the value from RBX is moved back to RAX (line 85), the RBX register is cleared (line 86), and the program jumps to the default return behaviour (line 87).

At the end of the function, the default return behaviour is shown again (lines 90–91). Here, the return value is directly stored in RAX, and the program returns to the caller. Comparing this behaviour to the return inside the try block reveals a clear difference. In the default case, there are no additional steps involving the RBX register or a jump to the finally block, demonstrating how the control flow is modified when a finally block is present in combination with a try-catch structure.

This example effectively showcases how the assembly semantics of try-catch-finally structures alter the default control flow to ensure proper handling of cleanup code in the finally block.

```
"test":
  push %rbp
  mov %rsp, %rbp
  sub $16, %rsp

  # TRY_BLOCK
  # RETURN
  mov $42, %rax
  mov %rax, %rbx
  jmp .L..0finally
  # CATCH_BLOCK
.L..0catch:
  mov %eax, -4(%rbp)
  # FINALLY_BLOCK
.L..0finally:
  test %rbx, %rbx
  je .L..0end
  mov %rbx, %rax
  xor %rbx, %rbx
  jmp 9f
.L..0end:
  # RETURN
  mov $10, %rax
  jmp 9f
9:
  mov %rbp, %rsp
  pop %rbp
  ret
```

Figure 2.3 Try-Catch-Finally Assembly Code Snippet

4. Discussion

The primary goal of this research, as outlined in the proposal, was to introduce structured exception handling to the C programming language through modifications to the widcc compiler. This involved implementing the try, catch, and finally constructs, supporting nested try-catch-finally blocks, and ensuring robust control flow and cleanup mechanisms. Additional objectives included the introduction of predefined exception structures with error codes and messages and enabling users to define custom exceptions. The proposal outlined a phased approach, with lower goals focused on basic syntax and control flow, middle goals adding exception handling without nested support, and higher goals targeting complete functionality with nested blocks and custom exception structures. Although the basic objectives were achieved, there were differences between what we planned and what was achieved, as well as some generalizations and limitations. Here we reflect critically on these aspects.

In the proposal, our original plan was to modify the widcc compiler due to its simplicity, small size, and support for experiments. Our high-end goal was to support nested try-catch-finally blocks with proper control flow containing predefined exception structures with error codes and messages, as well as the ability for users to define custom exceptions. For the middle-end goal we wanted to implement exception handling with no nested support and no exception structures, while the lower goal focused on the basic syntax for try, catch and finally ensuring that the control flow is not aborted if an error occurs. The error encountered in try blocks would be skipped and the control flow would always go through the catch and final block as there would be no control flow logic.

The original plan aimed to use setjmp and longjmp for non-local jumps to introduce exception handling, as these functions offer a straightforward way to manage control flow without extensive changes to the compiler backend. However, the integration of setjmp and longjmp in the code generation phase was unsuccessful, leaving this feature unimplemented. This limitation restricts exception propagation to within the same try-catch block, preventing their use across function calls or stack frames. As a result, the current implementation lacks support for handling exceptions in deeply nested calls, reducing modularity and limiting its effectiveness for complex error handling. Future work could address this by replicating setjmp and longjmp behaviour directly with assembly instructions in the code generation phase.

In addition, although the proposal contained predefined exception structures with error codes and messages, these were not implemented. Instead, the current implementation relies solely on user-defined types, which may require additional effort from developers to create standardized exception handling.

Eliminating these limitations in future iterations would further improve the robustness and usability of the exception handling mechanism. Future work will focus on addressing these shortcomings by implementing error propagation and predefined exception structures to enhance the system's utility and flexibility.

In addition to the implementation, we have conducted tests to ensure the correctness and robustness of the exception handling mechanism in widcc. The tests in subject evaluated various scenarios, including throwing and catching diverse data types (int, float, char*, char, struct, long int, double) and managing nested and looping try-catch-finally blocks. Each test case is designed to validate specific aspects of the implementation, such as: proper control flow, reliable cleanup in finally blocks, and effective resource management using malloc and free. Testing effective resource management was particularly important to verify the shared scope within try, catch, and finally blocks. Since resource cleanup was successfully executed across these blocks, it also demonstrated the correctness of the shared scope mechanism, ensuring that dynamically allocated memory and other resources remained accessible and properly handled throughout the entire exception handling process.

To verify reliability, Valgrind was used to detect memory leaks and ensure that dynamically allocated memory was correctly freed. GDB (GNU Debugger) was used to debug control flow and inspect stack and register states during exception handling. Breakpoints were strategically placed at critical points, such as entering and exiting try, catch, and finally blocks, to confirm that the generated code worked as expected. Compiler debug statements have been added to trace the parsing and code generation processes, allowing inspection of the Abstract Syntax Tree (AST) and the generated assembly. Additionally, registers and the stack were closely monitored to ensure proper control flow, stack integrity during nested exception handling, and accurate propagation of thrown values to catch blocks. A thorough review of the generated assembly confirmed that labels for try, catch, and finally blocks were unique and correctly referenced, and that thrown values were stored and loaded accurately based on their types. These extensive testing measures collectively ensured a strong and reliable implementation of exception handling in widcc.

5. Conclusion

This research aimed to explore the feasibility of adding a built-in exception handling mechanism to the C programming language. By addressing the limitations of existing error management practices, such as reliance on error codes and manual checks, this project contributes to improving code reliability and maintainability in C.

The proposed approach focused on extending an existing compiler, specifically widcc, to implement try, catch, and finally blocks. This method aligns with the research hypothesis that integrating exception handling as a native feature in C would provide developers with a structured and efficient way to manage errors without compromising the language's performance and simplicity.

Although challenges remain, such as supporting exception propagation across function calls and predefined exception structures, the findings demonstrate that exception handling in C is achievable. The success of this implementation may benefit applications in domains such as embedded systems and operating systems, where efficient and reliable error management is critical.

The deliverables include a functional prototype, detailed documentation, and a scientific paper. These contributions aim to enrich the academic discourse on programming language paradigms and inspire further advancements in the evolution of the C language.

Acknowledgements

We would like to express our sincere gratitude to our research supervisor, Dr. Peter Lammich, for their invaluable guidance, support, and constructive feedback throughout the course of this research. Their expertise and encouragement have been vital in shaping the direction of this work. We are also grateful for their patience and thoughtful suggestions, which significantly improved the quality of this paper.

Additionally, we would like to thank the faculty of the Academic Research Skills course at University of Twente for providing a strong academic foundation, and to Winnie Dankers for their insightful lectures, support, and encouragement. Their contributions have been key in deepening our understanding of the subject matter and guiding us throughout the research process.

Declarations

Microsoft Word autosuggestions and paraphrasing tools were used throughout the document to improve writing.

References

[1] Lee, P. A. (1983). Exception handling in C programs. Software: Practice and Experience, 13(4), 389-405.

[2] Dentato, R. (2023). Exceptional C. Dev Community. Retrieved from https://dev.to/rdentato/exceptionalc-1idl

[3] Mocalvo, G. (2016). Exceptions4C. GitHub. Retrieved from https://github.com/guillermocalvo/exceptions4c

[4] Cristian, F. (1987). Exception handling. IBM Thomas J. Watson Research Division.

[5] GeeksforGeeks. (n.d.). Error Handling in C. Retrieved from https://www.geeksforgeeks.org/error-handling-in-c/

[6] LLVM Project. (n.d.). Clang: A C language family front-end for LLVM GitHub. Retrieved from https://github.com/llvm/llvm-project

[7] fuhsnn. (n.d.). WidCC: A simple C compiler for x86-64 Linux. GitHub. Retrieved, from https://github.com/fuhsnn/widcc

[8] [8] Microsoft. (2021). MSVC: C compiler. Microsoft. Retrieved from https://learn.microsoft.com/en-us/cpp/cpp/try-except-statement?view=msvc-170