Project Summary

This thesis investigates how memory safety can be enforced in C++ by restricting the language to a safe subset and applying static analysis. Modern C++ provides features and guidelines that encourage safer programming practices, but these are optional and not enforced by compilers. As a result, developers can still write code that compiles but violates safety principles, such as accessing invalid memory. This project aims to mitigate this by statically enforcing memory safety through a tool-supported analysis.

The proposed solution introduces a safe subset of C++, which disallows unsafe constructs like pointer dereferencing, `new` and `delete`, union field access, and certain type casts. Instead, it encourages the use of lvalue-references and smart pointers to manage resource ownership and lifetimes. This approach echoes the *Resource Acquisition Is Initialization* principle already prevalent in idiomatic and modern C++. By limiting language features in this way, the project makes it easier to perform safety analysis.

To enforce safety, a static analysis algorithm called *Safety Analysis* was developed. It ensures two main properties: that resources are not used after being moved or deallocated (lifetime analysis), and that accesses to resources are mutually exclusive (borrow checking). These checks are performed using control-flow and an auxiliary pointer analysis, and executed on a Control-Flow Graph.

The tool implementing *Safety Analysis* is called *CPlusPlusty*. It is built as a Clang plugin and can be used with existing C++ code bases compiled with Clang, without changing the language. Violations are detected early in the compilation process, preventing unsafe code from compiling. Developers can annotate types and functions to guide *CPlusPlusty*, similar to how Rust uses the `unsafe` keyword to isolate code that requires manual reasoning. *Safety Analysis* and *CPlusPlusty* are conservative and may disallow safe code, but this ensures that no unsafe code is erroneously accepted.

CPlusPlusty was evaluated through a series of small test programs and a larger real-world C++ project. The smaller programs were used to validate that *CPlusPlusty* correctly enforces the safe subset and detects violations from *Safety Analysis*, including some over-approximations like partial moves. The tool was also applied to a larger code base originally implemented in both C++ and Rust. With minor changes, the C++ version was made compatible with *CPlusPlusty*, demonstrating the feasibility of enforcing safety in practical applications.

Compared to other efforts such as 'Safe C++', `Cpp2`, and library-based borrow checkers, this project differs by not adding to the language or requiring a new frontend. Instead, it restricts the use of unsafe constructs and performs analysis directly on standard C++.

While *Safety Analysis* currently over-approximates and does not support polymorphism or exceptions, it lays a strong foundation for bringing Rust-like safety to C++. Future work includes refining *Safety Analysis* to handle more language features, improving precision with flow- and context-sensitive pointer analysis, and expanding testing to ensure scalability. This thesis demonstrates that memory safety can be enforced in C++ without changing the language itself, using tools that analyze and constrain standard, modern C++ code.

Enforcing Memory Safety in Modern C++ Through a Safe Subset and Static Analysis

Crust++

Department of Computer Science

CS-25-DS-10-05 , Spring 2025

Master's thesis

Title:
Enforcing Memory Safety in Modern C++ Through a Safe Subset and Static Analysis

Theme:
Crust++

Project Period:
Spring 2025

Project Group:
cs-25-ds-10-05

Participant:
Thomas Krogh Lohse

Supervisors:
Danny Bøgsted Poulsen
René Rydhof Hansen

Page Numbers:
21

Date of Completion:
2025-05-30

The Technical Faculty of IT and Design
Aalborg University
https://www.aau.dk

Abstract:

This thesis investigates how memory safety can be enforced in C++ by defining a conservative, safe subset of the language and applying static analysis. While modern C++ offers features and guidelines that support safer programming practices, these are not enforced by compilers and remain optional.

To address this, the project introduces a static analysis, that detects lifetime violations and enforces mutual exclusive access to resources, similar to Rust's borrow checker and lifetime model. This analysis is accompanied by a proof-of-concept implementation, *CPlusPlusty*, as a Clang plugin.

The approach restricts unsafe constructs by default and uses an over-approximating analysis to ensure soundness. *CPlusPlusty* demonstrates that memory safety guarantees can be retrofitted onto modern C++ code. The results suggest that such guarantees can be achieved without major language extensions or alternative frontends, merely by applying static analysis and enforcing safer usage patterns within standard C++.

AALBORG UNIVERSITY
STUDENT REPORT

Preface

This report is written by me, group cs-25-ds-10-05, as part of my master's thesis in the masters degree program in Software Engineering at Aalborg University. The central theme of the thesis revolves around "Crust++".

I express my gratitude to my supervisors, Danny Bøgsted Poulsen and René Rydhof Hansen, for providing valuable guidance and support throughout the semester.

Contents

1. Introduction

As concluded in [1], C++ can in fact be a safe language. However, due to the size and many constructs it has acquired over the years, some of which are unsafe, only a subset of C++ can actually be qualified as safe. As Bjarne Stroustrup, creator of C++, has stated: *"Inside C++, there is a much smaller and cleaner language struggling to get out."* [2]. In [3], he also states that developers should use modern C++, as that is safe an valid. While, he is definitely correct, that using modern constructs and patterns will lead to memory safe code, the fact the language does not force or even really encourage this, makes the language unsafe.

Following the guidelines [4] will make your C++ code modern, and safe. However, that is just a guide, not anything a C++ compiler will ensure by default, and following it will still not give the same safety guarantees as Rust gives. Take the example in Listing 1.1, this is obviously invalid and dangerous code, but allowed in any C++ compiler. In Rust, this would be caught by the borrow checker, because of their mutual exclusive access rule.

As described in [1], the U.S. government are explicitly demanding that their suppliers switch from C/C++ to a memory safe language. No doubt that more companies and organizations will demand their suppliers to move away from C++, and use a safe language, such as Rust, instead. Therefore, some changes to C++ are needed to make arguments for the usage C++ in the future. The example in Listing 1.1 illustrates that the solution is not just to prohibit pointers and other unsafe constructs, but to add analysis to the language, regarding resources.

Rust has two main differences compared to C++ when discussing memory safety: lifetime of references, and no data races. Lifetime of references ensures no use after freeing, no dangling references, and no use after move. Rust ensures no data races by executing their borrow checker that ensures mutually exclusive access to resources. These two aspects are not present in any possible subset of C++, meaning these analyses should be added, to match the safety guarantees of Rust.

This project will tackle this exact issue. Throughout this report, I will define a conservative, modern, and safe subset of C++, followed by an algorithm called *Safety Analysis*, designed to achieve the same two safety guarantees previously mentioned. The defined subset and analysis will have an accompanying proof-of-concept implementation, called *CPlusPlusty*. The result will be a new, modern C++, that achieves the same guarantees as Rust does, with some over approximations. This will, however, require a new way to program in C++, a way much like Rust.

```
1 void f()
2 {
3     std::vector<int> v(10);
4     int& p = v[5];
5     v.push_back(99); // could reallocate v's elements
6     int x = p;       // BAD: potentially invalid reference
7 }
```

Listing 1.1: Invalid reference, modified version of example from [4, ES.65].

1.1 Related Work

This section will dive into other projects, that, like this project, aims at making C++ a safer language.

1.1.1 Borrow Checker As a Library

While this project defines *Safety Analysis*, that contains a borrow checker-like analysis, and an implementation of it, C++ developers may not appreciate the language changing and only want to have the analysis done on explicitly stated parts. This could for example be done through a library. GitHub user Jaysmito101 has made a library called `rusty`, which provides a borrow checker to C++, along with some Rust-like types such as `Option` and `Result`. The drawback of this approach is that everything you want to have analyzed must be wrapped in the provided type `Val`, which may prove too cumbersome and divert developers from using it. Additionally, if it is easier to write unsafe code than safe, there will most likely be more unsafe code than vice versa.

1.1.2 Safe C++

'Safe C++' is a proposal (currently draft) from Sean Baxter, scientific programmer and author of Circle C++[1], and Christian Mazakas, software engineer at *The C++ Alliance*, about a safer C++ [5]. Like this project, the goal of 'Safe C++' is to introduce a borrow checker into C++, and guarantee memory safety at compile time. Their approach to this is through a safe, opt-in environment, `safe`, the compiler can guarantee no undefined behavior. Where this differs from this project is the scale and mindset. 'Safe C++' wants to add elements to C++ (additional reference type, with lifetime annotation) and make safety opt-in, whereas this project tries to restrict the language to its safe subset, and make safety opt-out.

'Safe C++' feels like a new, safe C++ frontend, whereas this project aims at solving the same problem, but within C++.

1.1.3 Cpp2 & Cppfront

Herb Sutter, convener in the C++ committee, has created a new frontend for C++ called `Cpp2`, along with a compiler for it called `Cppfront`[2]. In his own words, this project is a way to *"make writing ordinary C++ types/functions/objects be much simpler and safer, without breaking backward compatibility"*. Besides the new syntax, we now get a bunch of safety improvements, such as bounds and null checks injected and type correctness requirements in operations, along with some unchecked functions (functions named with an `unchecked` prefix) for performance.

This project suffers from the same problems as 'Safe C++'; this is a frontend solving a problem that is solvable from C++ itself.

These projects either expands C++, or creates something beyond it. This project differs in the approach, mainly that it solves the problem by restricting the language, and analyzing that.

[1] https://www.circle-lang.org/site/index.html – Accessed 16. January 2025
[2] https://hsutter.github.io/cppfront/welcome/overview/ – Accessed 5. December 2024

2. The Safe Subset

This chapter outlines the subset *Safety Analysis* will consider. First, some argumentation for how I deem it best to re-design an unsafe language to only allow the safe parts (by default). Then, a short description of the subset, along with what has been left out for now. The main concern with this project is memory safety, specifically for pointing datatypes, meaning the created subset may not be exhaustive.

C++ has been around for close to half a century, while being actively maintained and expanded upon. This has the obvious effect that it has grown to be an extremely large language, consisting of a lot of new constructs with modern (and arguably better) practices, that coexists with the older and (potentially) unsafe constructs.

As concluded in [1], C++ has the capability to be a memory safe language, however, it needs to restrict itself to the safe, modern constructs and disallow the unsafe constructs by default, allowing to use those only through an opt-in approach. Identifying what is and is not included in safe subset, along with deciding on potentially contradicting features to include, can be a difficult task, and would most likely require a lot of studying and understanding of the language. The approach taken in this project is to start with a potentially overly conservative subset, allowing for expansion. With this conservative subset, we can apply some safety analyses, and from there expand the language by including elements in said analyses.

This is the idea used throughout the report, apply conservative analysis that may over approximate, but can be extended/modified to be less conservative, and/or include additional constructs from the language.

2.1 Removals and Disregards

[1] describes some unsafe constructs in C++, along with what the difference between safe and unsafe Rust is. The restrictions for the same subset mirrors that of Rust's, and the following is therefore prohibited in the safe subset of C++:

- Dereferencing of pointers.
- Accessing fields in `union`s.
- `new`- and `delete`-expressions.
- Use of labels and `goto`.
- `reinterpret_cast`-expressions.
- `const_cast`-expressions.
- C-style cast expressions

Creating pointers is still allowed, mainly for using unsafe *Application Programming Interface*s (APIs), but utilizing them is prohibited. This is because, unlike with lvalue-references, pointers can represent so much. Pointers can be constructed from any arbitrary integer (by casting), can be `nullptr`, can more easily point to deallocated data, and finally, pointers may or may not own the data it points to, without any way to convey this other than the documentation of the API. Following this argumentation, constructing `union`s is still allowed, but accessing them is prohibited, since `union`s are inherently type unsafe.

`new`- and `delete`-expressions are prohibited, because they create/delete an owning pointer, which should only be done through smart pointers. If using an API that takes an owning, raw pointer, that should be done through the `std::unique_ptr::release` method (if the API is deallocating it with `free`, a safe `malloc`-wrapper smart pointer is needed). If receiving an owning, raw pointer, that should immediately be wrapped in a smart pointer.

```
struct Obj {
    virtual AnyType &method(...) = 0;
}

void some_function(Obj &v) {
    auto &value = v.method();
}
```

Listing 2.1: Example of pure `virtual` function with an lvalue-reference return type.

Using labels and (explicit) `goto`s are also removed, since these can be used to move the execution to some arbitrary point in the control flow. However, the safe variants (`return`, `break`, `continue`) are still allowed.

The three potentially unsafe casting-expressions are prohibited. Notably, `static_cast`- and `dynamic_cast`-expressions are still allowed. These are considered safe, though `static_cast` can be unsafe in polymorphic contexts, if not used with caution. However, polymorphism is not considered in *Safety Analysis*, making `static_cast` perfectly safe.

Polymorphism (specifically subtype polymorphism through class inheritance) is not considered in this analysis to follow the method described above with starting small and expanding, and because pure `virtual` functions returning lvalue-references will demand too much to be analyzed. Take the example in Listing 2.1, where `method` may take any number of arguments of any type, and returns some lvalue-reference, it's impossible to know whether the returned value points to some field on the type, a function argument, a global object, or something it allocates on the stack. Currently, there is no way for developers to specify this, whereas Rust uses lifetimes to accomplish this. Either, some assumptions would have to be made and verified with regards to what object the returned lvalue-reference is associated with, or developers should have to specify this. Either way, these are out of the scope of this project and is considered future work. Abstract classes and `virtual` functions are not prohibited, but *Safety Analysis* does not consider them. Additionally, exceptions are also not considered (or more specifically, not prohibited) in the safe subset, along with lambda expressions.

2.2 The Safe Subset

The actual safe subset would consist of the remaining of the C++ language. Considering references/pointers in this subset, there are only two: lvalue-references and smart pointers. Lvalue-references are non-owning references to some value, and smart pointers are owning pointers, storing the data on the heap. This follows the *Resource Acquisition Is Initialization* (RAII) principle by default, since smart pointers ensures deallocating the data when it is no longer used, and lvalue-references cannot own what they point to.

With this model, dynamically-allocated resources can only be created through smart pointers, and using those resources through lvalue-references, can only be done by dereferencing the smart pointer.

3. Analysis Design

This chapter outlines and explains the analysis, called *Safety Analysis*, that is used to achieve memory safety for the safe subset of C++. The goal of *Safety Analysis* is twofold: Ensure that all resources (and references to them) are always live, meaning the resource has not been moved or deallocated when used (Lifetime analysis), and ensure mutually exclusive access to all resources (Borrow checking). Mutually exclusive access is the same rule that Rust's borrow checker achieves.

The overall design of *Safety Analysis* is loosely inspired by [6]; the result from an auxiliary pointer analysis, **AUX**, is supplied to two dataflow analysis algorithms. The result from **AUX** represents owning and non-owning references/pointers, distinguished by the type of the variable, I.E. whether it's an lvalue-reference, or a smart pointer. The implementation created for this project will use an Andersen style pointer analysis [7], and is described in Chapter 4.

Safety Analysis is composed of two algorithms, which are described in the following sections. They are both are executed on the *Control-Flow Graph* (CFG). A CFG is a directed graph that represents the flow of a program [8]. It has an entry- and exit-node, and may be cyclic (due to loops). Each node has a set of preceding nodes, and a set of succeeding nodes. A node in a CFG represents a basic block, which is a linear sequence of instructions, with exactly one entry and one exit.

3.1 Lifetime

The lifetime check is two-fold: Check that a variable has not been moved before a usage, and check that all potential pointees has at least one live owner. A variable '`var`' is moved if it is used as the argument in a `std::move` function call (`std::move(var)`). The lifetime check is run on all uses of all variables.

The move-check involves going back up through the CFG to check whether a move exists before any potential re-assignment, in any preceding path. Re-assignments are only applicable for non-lvalue-references, because lvalue-references are bound to the data they initialized with, and cannot be re-assigned afterwards to reference another object [9], making them similar to *Single Static Assignment* (SSA) form [10, ch. 1.1].

The live-owner-check checks whether all pointees has not been moved, and has at least one live owner. This check is only done on non-parameter variables. This is because since we can only have lvalue-references and smart pointers. Lvalue-references can only be constructed from variables or by copying another lvalue-reference, meaning any invalid construction will be caught when calling the function, and if the parameter is a smart pointer, it will own the resource and is therefore valid. A resource can have multiple owners thanks to smart pointers (specifically `std::shared_ptr`), but only in cases where the pointee is a heap-allocated resource. In many cases, the owners will resolve to a set with a single element of a stack variable. The move-part of this check is like the move-check described previously. The live-owner-part checks whether any of the owners are in the scope of the current CFG, and that they have not been moved.

The overall lifetime check can be generalized this formula:

$$\forall_{v \in VARS}.\forall_{s \in USES(v)}.\neg MOVED(v,s) \wedge LIVE\_OWNER(v,s)$$

$$MOVED(v,s) = \begin{cases} false & IS\_ASSIGNMENT(v,s) \\ true & IS\_MOVE(v,s) \\ \bigvee\limits_{s' \in PREDS(s)} MOVED(v,s') & otherwise \end{cases}$$

```
1  void copy_parm(int &p)
2  {
3      int &a = p;
4      a;
5  }
6
7  int main()
8  {
9      int vv = 10;
10     copy_parm(vv);
11 }
```

Listing 3.1: Parameter copy.

$$LIVE\_OWNER(v,s) = \neg IS\_PARM(v) \implies \forall_{p \in PTS(v)}.\neg MOVED(p,s) \wedge \exists_{o \in OWNERS(p)}.IS\_LIVE(o,s)$$
$$IS\_LIVE(v,s) = IS\_DECLARED\_IN\_SCOPE(v,s) \wedge \neg MOVED(v,s)$$

where *Safety Analysis* will fail if this is not satisfied.

3.1.1 Over approximations

This definition currently does not handle partial moves (moving a field). If a partial move is met, for example `std::move(var.field)`, this will be elevated to be a move of the variable itself, `std::move(var)`. This is an over approximation to ensure no false positives. Likewise, a partial assignment, for example `var.field = value`, will be ignored completely for the same reasons.

Additionally, the IS_LIVE check has an over approximation. Since it checks for owners within the same scope, lvalue-references created by copying a parameter will cause an error. Take the code in Listing 3.1, this code would produce an error because `a` points to `vv` because it is copied from `p`. And since `vv` is not in the scope of `copy_parm`, *Safety Analysis* would produce an error.

3.2 Borrow Checking

The borrow checking check is to ensure that mutable access to a resource is exclusive within its region, described in Section 3.2.1.

This check only considers mutable variables. If the variable is mutable, a check for exclusivity must be performed, comparing every potential shared pointer, acquired through $SHARED\_PTS$. $SHARED\_PTS$ returns every potential common pointer of the given resource. That is, if the variable v is an lvalue-reference, all other lvalue-references pointing to the same resource will be included in the result, and if it is not an lvalue-reference, all lvalue-references pointing to v will be included. We include nested cases, because it is still possible to have an lvalue-reference-field. Say, we have type `A` with field `int &p`, we can have an lvalue-reference to a variable of type `A`, where its `p`-field holds an lvalue-reference to v. Likewise, if v contains fields, and some other lvalue-reference points to that field, it will also be included.

The exclusivity check depends on the order of declarations. This is because, we want to ensure that no usages of the outer-most variable of the two being compared, is used in the inner-most variables region. Similar to the lifetime-check, this can also be expressed in a formula that must be satisfied for a safe program:

$$\forall_{v \in VARS}.MUTABLE(v) \implies \forall_{v' \in SHARED\_PTS(v)}.\begin{cases} skip & v = v' \\ EXCLUSIVE(v,v') & DECL\_AFTER(v',v) \\ EXCLUSIVE(v',v) & DECL\_BEFORE(v',v) \end{cases}$$

$$EXCLUSIVE(INNER,OUTER) = \bigwedge_{u \in us} NEXT\_USES(u,INNER) = \emptyset$$
$$\text{where} \quad us = NEXT\_USES(DECL(INNER),OUTER)$$

```
int p = 10; // PUSH p -- stack: [p]
int &d = p; // PUSH d -- stack: [d, p]
int &x = d; // PUSH x -- stack: [x, d, p]
use(x);     // NO OP  -- stack: [x, d, p]
use(d);     // POP 1  -- stack: [d, p]
use(d);     // NO OP  -- stack: [d, p]
use(p);     // POP 1  -- stack: [p]
// No empty stack, meaning no error
```

(a) Stack borrowing success.

```
int p = 10; // PUSH p -- stack: [p]
int &d = p; // PUSH d -- stack: [d, p]
int &x = d; // PUSH x -- stack: [x, d, p]
use(d);     // POP 1  -- stack: [d, p]
use(d);     // NO OP  -- stack: [d, p]
use(p);     // POP 1  -- stack: [p]
use(x);     // POP n -- stack: []
// Error thrown: `POP`-op on empty stack
```

(b) Stack borrowing failure.

Listing 3.2: Stack borrowing examples.

```
std::vector<int> v{1};
int &p = v[0];

for (int i = 1; i <= 100; i++) {
    use(p);
    v.emplace_back(i*2);
}
```

Listing 3.3: Looping error.

$EXCLUSIVE$ checks that all of the next uses of the outer-most variable (after the inner-most variable's declaration), does not lead to any uses of the inner-most variable. $NEXT\_USES(p, VAR)$ will search the succeeding paths after program point p and return the first use(s) of VAR. If $DECL(VAR)$ is met, make an early return with $\emptyset$. This is because, that would indicate a loop, where the declaration of VAR is within the loop.

This check creates a similar concept as *stacked borrows* (mentioned in [11], formalized in [12]), where the idea is that accesses to resources happens through a stack. If a declaring a variable `v` to point to `p`, it is the only way to access the resource, and when a usage of `p` is met, pop the stack until the peak of the stack is `p`. If it ends up popping the entire stack, it means that there is no exclusive access. Listing 3.2 illustrates this.

3.2.1 Regions

A region is all statements from a variable declaration and its uses, to every use of the same variable, without the declaration between. This definition is based on the paths on a CFG and will, ensure that the entirety of a variables usage, is exclusive. The reason for this definition, is to ensure exclusive access within loops as well. Take the code in Listing 3.3, here `p` is pointing to the first element of `v`. In the loop, `v` is expanded, which may (in this exacts scenario, will) demand the vector to expand, and therefore re-allocate memory. If this happens, and another iteration of the loop is executed, `p` will point to de-allocated memory.

This definition is an over approximation for non-lvalue-reference variables, since they can be re-assigned to another value. However, because **AUX** has no restrictions regarding its precision (described in detail in Section 3.3), the definition of a region must be over approximating. Generally, region can be viewed as a variable's lifetime from Rust, but over approximating, except code in SSA-form (and by extension, lvalue-references).

3.3 Precision of AUX

The more precise the **AUX**'s algorithm is, the more precise *Safety Analysis* is. For example, using a context-insensitive analysis will yield a less precise result than using a context-sensitive algorithm. Take the example in Listing 3.5. Here, a context-insensitive pointer analysis may result in $\{\texttt{ap} \to \{\texttt{a}, \texttt{b}\}, \texttt{bp} \to \{\texttt{a}, \texttt{b}\}\}$, because it merges all function calls, whereas a context-sensitive will result in $\{\texttt{ap} \to \{\texttt{a}\}, \texttt{bp} \to \{\texttt{b}\}\}$, since it utilizes the context of each call. With this example, a borrow checker would fail if the region from first and last use of `ap` and `bp` were completely disjoint or encapsulated, if it used the context-insensitive result. This is because a borrow checker needs exclusive access in the region a mutable variable is used. However, using the context-sensitive

```
1 int a;       // Region start -+
2 ....         //               |
3 ....         //               |
4 ....         //               |
5 a;           // Region end ---+
6 ....
7 ....
```

(a) Basic region. One declaration, one use.

```
1  int a;       // Region start -+
2  ....         //               |
3  if (cond) { //               |
4      ....     //               |
5      a;       // Region end ---+
6      ....     //
7  } else {     //               +
8      ....     //               |
9      a;       // Region end ---+
10     ....
11 }
```

(b) Branch region. One declaration, two uses.

```
1 int a;          // Region start -+
2 ....            //               |
3 while (cond) { //               |
4     ....        //               |
5     a;          //               |
6     ....        //               |
7     ....        // Region end ---+
8 }
```

(c) Loop region. One declaration, one use.

Listing 3.4: Region examples.

```
1  void use(int &) { }
2  int &id(int &i) { return i; }
3
4  int main() {
5      int a = 10;
6      int b = 20;
7      int &ap = id(a);
8      int &bp = id(b);
9      use(ap);
10 }
```

Listing 3.5: `id`-function and usage.

result, it would know that `ap` and `bp` will not point to the same value.

Using a context-insensitive pointer analysis may cause some over-approximated errors in code. For example, in Listing 3.5, because both `ap` and `bp` may point to the same while being mutable, `ap` needs exclusive access in the region from its declaration to last use. But because we use the one of the potential pointees of `ap` between its declaration and usage, *Safety Analysis* will fail.

Likewise, using a flow-sensitive analysis will yield more precise results than using a flow-insensitive, because re-assignments are handled instead of merged. With a flow-insensitive analysis, a resource would require exclusive access from declaration to final use for all the resources it can potentially point to, whereas with a flow-sensitive, it would only be required from each assignment to the last use before the next.

However, because of lvalue-references are SSA-like, and because a flow-insensitive analysis executed on code in SSA form produces a flow-sensitive result, the gains from a flow-sensitive analysis are only applicable to smart pointers. This means that if smart pointers are frequently re-assigned, this more precise analysis will be beneficial, however, if they are not, there is not much to gain from using a flow-sensitive **AUX**.

Both context- and flow-sensitive **AUX** will most likely require some results-formatting that may affect a concrete implementation for *Safety Analysis*. A context- and flow-insensitive would simply map a variable to all a conservative points to set. However context-sensitive (call-sensitive) result, would map the variable along with the context to a points-to set, and a flow-sensitive would map a variable and program point to a points-to set. These would require some handling when traversing the CFG.

4. Implementation

This chapter describes the proof-of-concept implementation of *Safety Analysis* described in Chapter 3, called *CPlusPlusty*. First, Clang plugins will be described, as this is what *CPlusPlusty* is implemented as, which allows it to be used with the Clang compiler on existing C++ code. Then, the concrete implementation will be described. *CPlusPlusty* uses the Andersen pointer analysis [7] as **AUX**.

4.1 Clang Plugin

Clang plugins[1] are dynamic libraries that Clang can utilize to run code transformations and/or analysis during the compilation process. There are multiple ways to execute code transformation and/or analysis through Clangs ecosystem, for example using Clang Tidy[2], Static Analyzers[3], and Clang LibTooling[4], all with their pros and cons. Clang Plugin was preferred over the other options, because of their reasons specified[5], specifically that I want to make or break a build, and I want full control over the *Abstract Syntax Tree* (AST).

Working with Clang Plugins, you are mainly working on the Clang AST[6], but have access to much of the Clang (and LLVM) library, such as their internal, highly optimized data structures, attributes parsing and creation, and their CFG representation.

Working with the AST, there are two approaches: Using their `RecursiveASTVisitor` class[7], or AST matchers[8]. `RecursiveASTVisitor` is a class you can derive from and override the relevant methods for visiting the relevant parts of the AST, but can make extracting specific patterns more cumbersome, partially due to the size and complexity of the AST Clang provides, and C++ as a whole. AST matchers are effectively composite functions that match on patterns in the AST. They are used extensively in Clang Tidy checks, and does not require much to match on the needed patterns in the AST, but each matcher is disjoint meaning shared data/behavior may require them to be defined within the same abstraction. For this implementation, both are actually used, however, `RecursiveASTVisitor` is the main part of the pointer analysis implemented. `RecursiveASTVisitor` was chosen because the documentation for Clang Plugins (and FrontendActions) were exclusively using `RecursiveASTVisitor`, and because `RecursiveASTVisitor` provides more intuitive control over the AST. That is not to say that one is better than the other, in fact, based on the documentation of AST matchers (and the extensive usage of them in Clang Tidy checks), it appears that they would be the best way to go, going forward. Github user `banach-space` has made a project with examples of multiple kinds of plugins[9]. This project has been a great help at showing how to make the fundamentals of *CPlusPlusty*, and getting this project started.

To register a plugin in the compilation pipeline, users need to supply Clang with the dynamic library by `-fplugin=/path/to/plugin` Clang plugins also can also parse commandline arguments, those are supplied by adding the arguments `-fplugin-arg-REGISTERED_PLUGIN_NAME-ARG`, which supplies the plugin registered with the name `REGISTERED_PLUGIN_NAME` with the argument `ARG`.

[1] https://intel.github.io/llvm/clang/ClangPlugins.html – Accessed 07. May 2025
[2] https://clang.llvm.org/extra/clang-tidy/ – Accessed 07. May 2025
[3] https://clang-analyzer.llvm.org/ – Accessed 07. May 2025
[4] https://intel.github.io/llvm/clang/LibTooling.html – Accessed 07. May 2025
[5] https://clang.llvm.org/docs/Tooling.html – Accessed 07. May 2025
[6] https://clang.llvm.org/docs/IntroductionToTheClangAST.html – Accessed 07. May 2025
[7] https://clang.llvm.org/docs/RAVFrontendAction.html – Accessed 07. May 2025
[8] https://clang.llvm.org/docs/LibASTMatchers.html – Accessed 07. May 2025
[9] https://github.com/banach-space/clang-tutor – Accessed 15. May 2025

4.2 CPlusPlusty

The implementation of *CPlusPlusty* effectively supplies two frontend actions combined into one solution, registered under the name `safety_analysis`: A registry for a new attribute on function- and type-declarations, and the safety analysis. The attributes supplied can be used to mimic how Rust's `unsafe` works, as well as easily defining your own smart pointers. The overall attribute given is called `safety`, and has a handful of specifications that can should applied:

- `none` – Omit analysis within the function this is attribute applied to.
- `smart_ptr` – Tell *CPlusPlusty* that the type the attribute is applied to is a smart pointer.
- `smart_factory` – Tell *CPlusPlusty* that the function this attribute is applied to, is a smart pointer factory (for example `make_unique`).
- `smart_clone` – Tell *CPlusPlusty* that the function this attribute is applied to, is a smart pointer cloning function (for example a copy-constructor).
- `smart_deref` – Tell *CPlusPlusty* that the function this attribute is applied to, is a dereference of a smart pointer, and returns the heap-allocated value.

Functions with any of these these attributes do not have *Safety Analysis* executed on them, as operations regarding smart pointers will most likely require some pointer dereferencing, and other unsafe operations. With these attributes, one can easily make a custom smart pointer by annotating their class with `safety(smart_ptr)`. For example, if my type `A` is a smart pointer, it should be attributed like `class [[safety(smart_ptr)]] A {...};` In the same fashion, a wrapper around a C-API may need to be annotated with `safety(none)` if pointer dereference is needed. `safety(smart_ptr)` can only be applied to record types, and the remaining attributes can only be applied to functions/methods. Any misuse will cause an error, and the program will not compile. Throughout visiting the AST, some predefined types and functions are annotated (such as `unique_ptr` and `make_unique`) with the relevant specifications from above. Based on the annotations, some declarations that are not annotated by the user, are also annotated by inference. For example, if a type is annotated with `safety(smart_ptr)`, and it overloads a dereference operator (`*`, `->`, and `[]`), those methods will get the `safety(smart_deref)` annotation. The same for copy-constructor and -assignment operator overloads, they will be annotated with `safety(smart_clone)`, and any other constructor will be annotated with `safety(smart_factory)`. If using a library, or any code that you cannot annotate yourself, *CPlusPlusty* provides a few commandline arguments to tell the tool what functions and namespaces to ignore (effectively applying a `safety(none)`, and a way to add additional smart pointers.

The safety analysis part of *CPlusPlusty*, is executed before the main action of the compiler, which is code generation, and after the inputted code has been parsed and analyzed by the default analyses. This means that the analysis can take advantage of the fact that whatever code it is supplied with, is indeed correct. For example, if a copy-constructor for a class is deleted, and it is used in the code, the analysis will not be executed due to other preceeding errors. The safety analysis visits a great number of different kinds of nodes in the AST, though the most relevant for pointer analysis would be `VarDecl` and `BinaryOperator`/`CXXOperatorCallExpr`, where the operator is an assignment. `VarDecl` is when a variable is declared (and usually initialized), if the variable being declared is re-assigned later, the node will be a `BinaryOperator` or `CXXOperatorCallExpr` (depending on the type) instead. From each of these, some constraints are generated for the expressions in each. If the left-hand side is an lvalue-reference, and it is its declaration, either a copy- or addr-constraint is generated. It will be a copy-constraint if the right-hand side is also an lvalue-reference (or a function that returns one), and it will be an addr-constraint if it is not. Later assignments will be store- or loadstore-constraint due to lvalue-references' SSA form, and instead updates the values in their pointees. For *Safety Analysis*, CFGs are generated for each function, and the formulas are checked for each of those. The analysis only considers trivially reachable basic blocks, meaning if a given block is breaking the analysis, but it is unreachable (for example the else-branch in an `if (true)`), the code will still compile.

CPlusPlusty follows the formulas show in Chapter 3, and outputs diagnostics for the variables that fails a formula, rather than a general error it the entire formula fails. For example, the code shown in Listing 4.1a will produce the error shown in Listing 4.1b. To use *CPlusPlusty*, simply use the command: `clang++ -fplugin=/path/to/plugin /path/to/file`, and the analysis will be executed, and only produce an executable if it passes.

```cpp
int main()
{
    int d = 10;

    int &p1 = d;
    int &p2 = d;
    p1;
}
```

(a) Failing code.

```
/path/to/file.cpp:5:5: error: Borrow checker
    5 |     int &p1 = d;
      |     ^~~
      |     Conflicting mutable access: `p1` and `d` have overlapping regions.
/path/to/file.cpp:6:5: note: Inner var use
    6 |     int &p2 = d;
      |     ^~~
      |     This use of `d`
/path/to/file.cpp:7:5: note: Outer var use
    7 |     p1;
      |     ^~
      |     Has path to use of `p1`
1 error generated.
```

(b) Diagnostic.

Listing 4.1: Erroneous code and its diagnostic.

4.2.1 Limitations

For this implementation, all types and functions within system headers (meaning the standard library) are ignored. This was done because the standard library has not been written with the safe subset described in Chapter 2 or *Safety Analysis* in mind, and would most likely cause several errors. If the C++ language adopted the subset and analysis described in this report (or some variant of it), the standard library would need some re-writing, or at least annotate the unsafe parts. Additionally, this implementation may, unknowingly, omit some edge cases from C++. This is only solvable by a very well-defined subset and analysis, along with a big test suite, testers, coders, and code reviewers to catch all cases.

This plugin completely disregards all functions annotated with any of the previously mentioned attributes, which is unlike Rusts `unsafe`. Rust still analyze code in `unsafe` blocks or functions, but just allows some operations to be performed. This way, error such as multiple mutable borrows and use after move are still caught. Ideally, *CPlusPlusty* should, like Rust, still analyze the annotated functions, but simply ignore the subset described in Chapter 2. This would, however, demand that all of the code being analyzed is written with the safe subset and *Safety Analysis* in mind, therefore still demanding system headers to be omitted or re-written.

5. Test

This chapter describes how *CPlusPlusty* has been tested. The testing is two-fold: A suite of small, toy programs to illustrate the different cases *CPlusPlusty* may encounter, and then *CPlusPlusty* executed on a larger program.

5.1 Test Suite

The test suite consists of over 60 small programs, each covering its own construct to show what *Safety Analysis* prohibits, and what it allows. The tests mostly contains under 50 lines to setup and execute the test, and the entire suite is executed through `ctest`. A small bash script gets the path to the file, and an expected status (success or failure), and executes the program, reporting if the test has failed or not. Some of the over approximations mentioned throughout the report are also tested, and, as expected, they do in fact over approximate.

The test suite was written with the subset in mind, however, it may miss a few cases with more exotic C++ code. For example, code with large amounts of templates and concepts has not been tested, along with variadic templates. These are cases where *CPlusPlusty* relies on Clang's AST. A test for the `id`-function shown in Listing 3.5 was made that shows it fails, but a test case with generics and differing types for `a` and `b` (along with their lvalue-reference variants) does compile, thanks to C++ monomorphism, and that the Clang AST clearly and easily differentiates those.

Something that should be considered is a more complex test suite, with auto-generated test cases, and perhaps some mutation testing. Implementing mutation testing could ensure that the program catches different cases, such as mutating an lvalue-reference to a pointer (and subsequently all uses of the variable as dereferences), and see if *CPlusPlusty* will catch the use of the unsafe constructs. The control Clang provides over the AST makes it easy to modify an input program. The procedure to ensure correct mutations just needs to be defined.

5.2 Larger Code Base

CPlusPlusty has been tested on the C++ program developed in [1][1]. This actually caught a few instances of a bad practice, namely allocating through `new` expressions. These were then saved in a `std::unique_ptr`, meaning one should simply use `std::make_unique` to allocate the required memory. After this fix, *CPlusPlusty* showed some over-approximation regarding the CFG. *CPlusPlusty* caused an error for the code shown in Listing 5.1. The error occurs because there is a path from the move of `engine` on line 2, to the usage of the variable on line 22. We as developers can clearly see that the only path to line 22, is the path where `engine` is re-assigned, however, without any additional analysis to mark some paths in the CFG as unreachable, there are multiple paths to line 22. This implementation was made to, stylistically, be similar to the Rust implementation[2] that uses `match`. A simple fix to ensure the analysis passes, is to omit the `switch`, and simply check the two error cases in two `if`s, follow by the re-assignment. This fix is shown in Listing 5.2.

Using *CPlusPlusty* initially, it was commanded to ignore all functions within the `gsl` namespace. This was done for the same reasons as *CPlusPlusty* by default ignores system headers, because the `gsl` library was not

[1] Pull request with the changes to make it compile can be found here: https://github.com/t-lohse/MicroProfileEngine/pull/2
[2] https://github.com/t-lohse/micro-profile-engine-rs

```
while (engine.getState() != ProfileState::Done) {
    auto newEng = std::move(engine).step();
    bool dip = false;
    switch (newEng) {
        using T = EngineStepResult<DummySensorState>;
        case T::Next:
            engine = std::move(newEng).getNext();
            break;
        case T::Finished:
            dip = true;
            break;
        case T::Error:
            std::cout << "No stages in profile!!! Error: `" << std::move(newEng).getError()
                      << "`" << std::endl;
            return 1;
    }
    if (dip)
        break;

    const long SLEEP_TIME = 50;
    std::this_thread::sleep_for(std::chrono::milliseconds(SLEEP_TIME));
    std::cout << "The engine is in state: " << engine.getState() << std::endl;
    if (engine.getState() == ProfileState::Brewing) {
        const double PISTON_CAP = 100;
        *pistonPos = std::min<double>(*pistonPos + 1, PISTON_CAP);
        std::cout << "Piston: " << *pistonPos << std::endl;
    }
}
```

Listing 5.1: Original code.

```
 while (engine.getState() != ProfileState::Done) {
    auto newEng = std::move(engine).step();
    using T = EngineStepResult<DummySensorState>;
    if (newEng == T::Finished) {
        break;
    } else if (newEng == T::Error) {
        std::cout << "No stages in profile!!! Error: `" << std::move(newEng).getError()
                  << "`" << std::endl;
        return 1;
    }
    engine = std::move(newEng).getNext();

    const long SLEEP_TIME = 50;
    std::this_thread::sleep_for(std::chrono::milliseconds(SLEEP_TIME));
    std::cout << "The engine is in state: " << engine.getState() << std::endl;
    if (engine.getState() == ProfileState::Brewing) {
        const double PISTON_CAP = 100;
        *pistonPos = std::min<double>(*pistonPos + 1, PISTON_CAP);
        std::cout << "Piston: " << *pistonPos << std::endl;
    }
}
```

Listing 5.2: New code.

written with the safe subset and *Safety Analysis* in mind, and would most likely cause some error. However, the library was not used that much, so the program was changed such that usages of `gsl` was removed or changed to lvalue-references. This new implementation still compiled with *CPlusPlusty*, and resulted in the same as before the changes.

6. Discussion

This chapter will discuss the safe subset of C++, *Safety Analysis*, and *CPlusPlusty*, mainly regarding shortcomings and changes to improve scalability.

6.1 The Subset

The subset currently prohibits dereferencing pointers as the only restriction on pointer operations, meaning pointer arithmetics is technically allowed. This generally echoes what Rust allows for pointers. However, we may not have to completely disregard pointers, because of the ergonomics of lvalue-references. Once an lvalue-reference has been created, it can never point to something else. This is different compared to references from Rust, that essentially are re-assignable lvalue-references, with some static analysis attached to them. This makes Rust somewhat more capable, and forces C++ developers using the safe subset and *Safety Analysis* to create code in SSA form (when using lvalue-references). In Rust, the simplicity of using and re-assigning references, along with the guarantees of a live pointee, makes using them feel like a very high level language without pointers, but with the performance of a lower level language with pointers. Expanding the subset to include pointers in some capacity may be desirable, if C++ should be more like Rust, or simply keeping it as is, and letting C++, and how to program in it, adapt. A clear distinguishment between a safe and unsafe pointing-construct makes code a lot easier to understand and comprehend. Rust's references are more ergonomic regarding their safety analysis, because they were designed to be safe, C++'s lvalue-references were not.

The subset currently does not consider both polymorphism and exceptions. Especially polymorphism, is crucial for strong, generic programming and some object oriented design patterns, such as the visitor pattern. The problem regarding polymorphism is that, as explained in Section 2.1, C++ either needs some sort lifetime annotation, or have some assumptions regarding returned lvalue-references and what they are allowed to bind to. If a lifetime-like annotation is needed, it may be wise to learn from Rust, and potentially use something different than an arbitrary lifetime (if a return value has lifetime `'a`, that notion is completely arbitrary and has no easily understandable meaning), and instead use a *origin*[1]. With this proposal, given a lvalue-reference return value, you specify what input or static variables it will have its lifetime associated with, making code more readable and understandable compared to Rust's lifetimes.

Regarding exceptions, using them is a taste preference. Inherently, they are not unsafe, however, they can make code both harder to read and understand, and considering them in analysis can be very difficult, since they create edges from every statement to the exit-node in a CFG. Given a language where exceptions is the default way to handle errors, compared to a more functional approach, utilizing APIs from the former may require more care than the latter. Take the example in Listing 6.1, here we see two comparable function signatures of a function for getting the first element of a vector. The C++ version has no information in the function signature, meaning developers are required to have knowledge regarding error handling from the function (meaning what exceptions it can throw), either through documentation, intuition, or inspection of the concrete implementation. In the Rust example, the information regarding the different ways the function can cause an error is, more or less, embedded in the signature (specifically the return type). Developers can see that the function may or may not return a value, and they have to handle the case where it cannot (or explicitly ignore), whereas with C++, you have to explicitly opt in for error handling, through `try-catch`. This is a very trivial example, but it is not hard to imagine more nuanced cases where there are multiple different kinds of failures. In Rust, those would be represented in a `Result`, where the error-type is either an enum or trait object, whereas in C++, it would

[1] https://smallcultfollowing.com/babysteps/blog/2024/03/04/borrow-checking-without-lifetimes/, section "Replacing a lifetime with an origin" – Accessed 15. May 2025

```cpp
template <typename T>
const T& head(const std::vector<T> vec&);
```

(a) C++ version

```rust
fn head<T>(vec: &Vec<T>) -> Option<&T>;
```

(b) Rust version

Listing 6.1: Example of `head`-API, meaning getting the first element of a vector.

```
1 struct A {
2     int x = 10;
3     int y = 20;
4 };
5 int main() {
6     A val{};
7     std::move(val.x); // invalidates `val`
8     use(val.y);       // causes error
9 }
```

Listing 6.2: Partial move over approximation example.

most likely be several different exceptions.

A suggestion may be to treat exceptions like panics from Rust or exceptions from Haskell, meaning not a part of the control-flow, but (generally) unrecoverable errors. With this treatment, analysis will be easier and APIs will be more clear, while code generally being more robust and reliable. The bad side regarding this, is that when actually recovering from these errors (like through `catch_unwind` in Rust), developers are responsible for the state of the program that caused the panic. [13, ch. 7.1] explains this well, demanding *exception safety* in unsafe code. A good example of missing exception safety is doing an operation that can cause a panic, while having an owning, not-smart pointer. This pointer will simply be removed, but the pointee will not be deallocated, causing a memory leak. The safe subset actually accomplishes this by disallowing `new` and other ways to allocate memory that is not held through a smart pointer, though, like with Rust's `Box`, there is a safe way to leak the data from a `std::unique_ptr`.

Regarding lambda expressions, a naive approach to include them would be to treat them as any lvalue-reference variable: the variables captured by reference in the capture list, would be in a lambda variables points-to set.

6.2 Safety Analysis

As stated in Section 3.1.1, *Safety Analysis* currently over approximates partial moves. This will invalidate parts of a data structure that is still valid to use. The effect is illustrated in Listing 6.2, where we can see that that moving a field will cause an error if using a separate field. Even if a re-assignment of `val.x` was in between line seven and eight, the over approximation demands a re-assignment of `val` for `val.y` to be valid. The desired effect would be for a move to cause the field moved, its subfields, and its super-components to be invalidated, until that, or a super-component, is reassigned. The section also mentions the over approximation regarding lvalue-references created by copying a parameter. While it would be unintuitive to have two lvalue-references to the same object within the same scope, there is no reason to prohibit it. Some re-work of the check should be done to ensure that this over approximation will not happen.

Safety Analysis currently work on the uses of variables, which will catch the potential errors. However, if a function is created that may cause an error, but is never used, that error is never caught. For example, the function shown in Listing 6.3 shows the function `get_zero` that returns an lvalue-reference from a function, where the reference is bounded to a stack variable, meaning the returned lvalue-reference is unusable. If line eight is uncommented, the analysis will cause an error, however, because `v` is never used in this version, this passes *Safety Analysis*, and compiles. The only way to catch these kinds of errors is if developers are testing their code. With this, it is possible for a badly written API to cause compile errors for the users code base.

While *Safety Analysis* has its flaws, it shows potential. It is possible to make static analysis of C++ code, that can catch the same errors as the Rust compiler, and *Safety Analysis* lays a good foundation for more precise analysis of C++ in the future.

```
int &get_zero()
{
    int a = 0;
    return a;
}
int main() {
    auto &v = get_zero();
    // v;
}
```

Listing 6.3: Bad function `get_zero`.

6.3 CPlusPlusty

CPlusPlusty is made as a proof-of-concept, and is not scalable. Regarding the pointer analysis implementation, [14] has found that storing constraints from Andersen's pointer analysis [7] in *Binary Decision Diagrams* (BDDs) can be very efficient memory-wise, and scalable to larger programs. This is echoed in [6], though for a flow-sensitive, context-insensitive pointer analysis. In *CPlusPlusty*, some expressions are going through multiple checks to analyze the characteristics of the expressions (if the expression is a plain variable, smart pointer dereference, function call, etc.), and some of these checks perform the same sub-checks, meaning duplicate checks. Optimizing these may improve performance. Additionally, some constraints may be produced twice, which adds to the first iteration of the worklist algorithm for the pointer analysis, though will not propagate through the rest. As mentioned in Section 4.2, the input code is actually traversed twice, once for the pointer analysis, and once for *Safety Analysis*. This could (and should) be optimized away to only need one traversal of the input. When performing analysis on the CFG, there may be some duplicate computations (for example, that one basic block leads to another). If memory allows for it, these results should be cached and used to improve the runtime of the compiler.

CPlusPlusty's **AUX** should be a more precise pointer analysis, specifically that it should be context-sensitive, and for maximum precision, flow-sensitive for smart pointers. Reasons for this is explained in detail in Section 3.3.

As mentioned in Section 4.2.1, system headers are not analyzed currently, since the standard library would require a re-write and/or review to ensure it conforms with *Safety Analysis*. Additionally, *CPlusPlusty* does not analyze functions annotated with any of the described attributes. This should be changed to behave much like Rusts `unsafe`, such that it only allows the users to use the prohibited parts of C++, but still analyzes the code.

The current implementation handles containers (`std::vector`, `std::set`, `std::map`, etc.) as smart pointers. This behavior is not wrong since these containers are smart pointers that allocate the resources on the heap. However, due to the implementation of the smart pointer operations, *CPlusPlusty* may over approximate with regards to containers. `safety(smart_clone)` annotated functions (or copy-constructors on `safety(smart_ptr)` types) will be treated like a `std::shared_ptr`, meaning it will be like two pointers, pointing to the same. This distinction is correct with the classical smart pointers, though not with containers as they perform a deep copy of the data, meaning they do not point to the same heap-allocated resource. A way to distinguish between classical smart pointers and containers is needed for full precision and no over approximation.

Range-for loops in C++ is syncatic sugar for iterating over a container. Allowing these in *CPlusPlusty* is quite useful, as these loops are used frequently (and recommended by the guidelines [4, ES.71]). However, the way they are expanded causes an over approximation for *CPlusPlusty*. An example is shown in Listing 6.4. This is because, in *CPlusPlusty*, `__begin1` has a "forced" copy-constraint with `vector`. And since `__begin1` is mutable (otherwise `++__begin1` would fail), *CPlusPlusty* views it as a mutable borrow. This means that, even when `vector` is `const` and `__begin1` cannot provide a mutable reference to anything in `vector`, if `vector` is present anywhere in `stmts`, there will be a borrow checker error, as the loop leads to another use of `__begin1`, without its decl in between. Therefore, the borrow checker will make an error on `vector` and `__begin1`, and not `i` since that has a declaration in the path.

[2] https://en.cppreference.com/w/cpp/language/range-for – Accessed 16. May 2025

```cpp
for (const auto &i : vector) {
    /* stmts */
}
```

(a) Range-for loop.

```cpp
auto &&__range1 = vector;
auto __end1 = __range1.end()
auto __begin1 = __range1.begin()
for (; __begin1 != __end1; ++__begin1) {
    const auto &i = *__begin1;
    /* stmts */
}
```

(b) Approximate expansion[2].

Listing 6.4: Range-for loop and its expansion.

6.4 Testing

Regarding testing *CPlusPlusty*, a test suite was created to check the edge cases, and *CPlusPlusty* was executed on a larger project. Regarding the test suite, one can never have too many tests, and having to explicitly define all tests can be a limiting factor. As mentioned in Section 5.1, adding some generation of test cases, and/or mutation testing, can be beneficial, in combination with a lot of explicitly defined edge cases, may catch outlying cases, since every test suite may be unique. If this was implemented, one would only need to report the programs that cause an error, fix the error, and use said program as a part of the explicitly defined test suite.

For larger programs, *CPlusPlusty* was tested against a single program, that was written as a C++ variant of a Rust program. *CPlusPlusty* accepts the program, after a few small changes. However, a single, larger code base does not necessarily prove correctness. Ideally, a suite of larger projects will help testing the scalability of *CPlusPlusty*, as well as correctness on code that most likely is a combination of all the cases from the test suite.

The issue in the code shown in Listing 4.1, is likely an effect of creating code stylistically similar to Rust, something C++ was not designed to be. The more native way to fix this in C++ is with labels and `goto`s. Instead of having a variable to check if the flow should `break` out of the loop, it would be much simpler to have a label after the loop, and a `goto` within the `case T::Finished:`-block, as well as making the analysis work. But, in the subset defined in this project, labels and `goto`s are prohibited. This leads to the question whether the safe subset is too restrictive, and maybe it should allow labels and `goto`s in some capacity. Otherwise, it could also be solved if C++ had labeled-loops, that you could `break` from, meaning the statement `break outer_loop;` would break out of the loop with the `outer_loop` label, no matter where the statement is. If the language decides to change even further by adding tagged unions and pattern matching, the code could be almost exactly like its Rust counterpart that uses a `match` expression.

7. Conclusion

Chapter 1 presented the issue of safety in C++, and that it has the capability to be safe, though only a subset of it. C++ allows for a lot of code that [4] will call bad practice and unsafe, but compilers will at best warn for such scenarios. Even if developers restrict themselves to the safe subset, use-after-free and other errors are still possible. Therefore, C++ needs to make the unsafe parts hidden and opt-in, and add some safety analysis. There exists other projects aiming to solve the same problem, however, none of those are solving it by restricting C++, but instead expanding or recreating it.

In Chapter 2, a conservative subset was defined to enforce safe, modern C++. Here, some language constructs was prohibited, such as dereference of pointers and some casting-expressions. Though some safe and modern features has been omitted from this project, such as exceptions and polymorphism, mainly to start with a small, conservative, and expandable subset.

Chapter 3 outlined the algorithm *Safety Analysis*, which is an algorithm comprised of two dataflow analysis algorithms: a lifetime analysis, and a borrow checker-like analysis. *Safety Analysis* uses the result of an auxiliary pointer analysis, **AUX**, to have knowledge of the ownership of the variables, along with ensuring mutual exclusivity.

Chapter 4 presents the implementation of *Safety Analysis*, called *CPlusPlusty*. *CPlusPlusty* is implemented as a Clang Plugin, meaning it can easily be used on existing code, along with utilizing existing, state-of-the-art constructs (such as their large AST and CFG) from a real-world compiler. *CPlusPlusty* uses Andersens pointer analysis [7] as its **AUX**, and provides some attributes for users to specify code to avoid analysis (corresponding to `unsafe` in Rust), and to specify smart pointers and some operations regarding them. With *CPlusPlusty*, it is possible to test and execute *Safety Analysis* on existing C++.

In Chapter 5, test of *CPlusPlusty* is outlined, presenting two aspects: the test suite of over 60 programs, and a test on a larger project. *CPlusPlusty* behaves as expected, and found a few flaws with the larger project, that were easily fixed.

Chapter 6 discusses aspects of the project that are lacking, and leaves some aspects to be desired. For the safe subset, inclusion of polymorphism is a must for the future. For *Safety Analysis*, some over approximations should be fixed to provide the most precise analysis possible. For *CPlusPlusty*, some performance optimizations and a more precise **AUX** is needed to make it more precise and scalable. And regarding the testing, generation of tests and mutation testing can prove beneficial, along with testing on additional larger projects, will provide more confidence of the correctness of *CPlusPlusty*.

This thesis defined a conservative safe subset of C++, and introduced a static analysis, *Safety Analysis*, along with a proof of concept tool, *CPlusPlusty*, capable of enforcing key memory safety guarantees. Inspired by Rust's borrow checker and lifetimes, the analysis statically prevents dangling references and ensures mutually exclusive access, achieving many of the same safe guarantees as Rust has, at compile-time. With this, it is proved that C++ in fact has the capability to be as safe as Rust.

Compared to related work, this approach requires no new language frontend or compiler. Instead, it operates directly on standard C++ code, restricting unsafe constructs and enabling safety as a default, opt-out model. While the analysis currently over-approximates certain behaviors and excludes polymorphism and exceptions, it demonstrates a promising direction for fitting memory safety into the future of C++.

7.1 Future Work

While this project proves that C++ can in fact have additional safety analysis performed on it, there are several aspects left to be desired. Many of the points from Chapter 6 should be considered implemented. For the safe subset, extensive research and argumentation should be done to expand the subset to include as much of current C++ as possible. The main points to take away are whether pointers are well-fitted, and how to include polymorphism, whether some assumptions should be made regarding `virtual` methods, or whether to introduce some lifetime-like construct to the language.

Extending *Safety Analysis* to include partial moves and assignments and not over approximate, as described in Section 6.2 should be done. To accommodate this, a naive approach would be to add an additional check called $PARTIAL\_MOVE$ that, for each field in a variable (included nested fields), checks if it is moved. Another approach would be to extend how a variable and its uses are embedded, such that we register specific fields, meaning $VARS$ includes member expressions like `var.nested.field`. That way, we could use more or less the same definition, only including checks for the parent variable(s) additionally in the $IS\_ASSIGNMENT$ and IS_MOVE cases, like:

$$MOVED(v,s) = \begin{cases} false & \exists_{v' \in COMPONENTS(v)}.IS\_ASSIGNMENT(v',s) \\ true & \exists_{v' \in COMPONENTS(v)}.IS\_MOVE(v',s) \\ \bigvee_{s' \in PREDS(s)} MOVED(v,s') & otherwise \end{cases}$$

where $COMPONENTS$ returns the set of the identifiers/components used in the member expression (giving `var.nested.field` would return {`var`, `nested`, `field`}). The error caused by copying an lvalue-reference parameter should also be fixed, as stated in Section 6.2. This fix would involve re-working the analysis, or include some additional condition that checks if an lvalue-reference was not constructed by copying a parameter.

Likewise, *CPlusPlusty* should go through iterations of improvement performance-wise to increase the scalability of it, along with a more precise **AUX**. Some points regarding this has been discussed in Section 6.3.

Additionally, Rust claims to be *Thread Safe* [15], meaning multi-threaded applications are perfectly safe. Rust attributes a lot of their thread safety to their type system and ownership model [16, ch. 16][15], which is something *Safety Analysis* mimics. Assuming the previous future work has been implemented regarding a stronger analysis, C++ can most likely also claim to thread safe, something Herb Sutter has stated they also want to work on, but memory safety takes priority [17] The missing part is to tell what types are thread safe, meaning what types can be sent and synced between threads. Introduction of a 'thread safe type'-notion demands changes to the C++ language (more specifically, some additions). Rust currently has the traits `Send` and `Sync`, but C++ does not have a trait-like type system, meaning users would have to specialize some type (much like with `std::hash`) to tell whether a type is safely sent over threads. Another thought regarding this could be to extend the types of constructors for classes to include 'thread safe constructor' (or 'send constructor' and 'sync constructor').

Reference

[1] Thomas Krogh Lohse. *Towards a Safe C++: Crust++*. Semester project, Aalborg University. 2024.

[2] Bjarne Stroustrup. *The design and evolution of C++*. Boston, MA: Addison Wesley, Mar. 1994.

[3] Bjarne Stroustrup. *21st Century C++*. Communications of the ACM Blog. Feb. 4, 2025. URL: https://cacm.acm.org/blogcacm/21st-century-c/.

[4] Bjarne Stroustrup and Herb Sutter. *C++ Core Guidelines*. [Online; accessed 10. Oct. 2024]. Oct. 2024. URL: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.

[5] Sean Baxter and Christian Mazakas. *Safe C++*. [Online; accessed 5. Dec. 2024]. Oct. 2024. URL: https://safecpp.org/draft.html.

[6] Ben Hardekopf and Calvin Lin. "Flow-sensitive pointer analysis for millions of lines of code". In: *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. IEEE Computer Society, 2011, pp. 289–298. DOI: 10.1109/CGO.2011.5764696. URL: https://doi.org/10.1109/CGO.2011.5764696.

[7] Lars Ole Andersen. "Program Analysis and Specialization for the C Programming Language". DIKU Research Report 94/19. PhD thesis. Copenhagen, Denmark: DIKU, University of Copenhagen, May 1994. URL: https://www.cs.cornell.edu/courses/cs711/2005fa/papers/andersen-thesis94.pdf.

[8] Frances E. Allen. "Control flow analysis". In: *Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970*. Ed. by Robert S. Northcote. ACM, 1970, pp. 1–19. DOI: 10.1145/800028.808479. URL: https://doi.org/10.1145/800028.808479.

[9] Standard C++ Foundation. *References, C++ FAQ*. [Online; accessed 15. May 2025]. May 2025. URL: https://isocpp.org/wiki/faq/references.

[10] Florent Bouchez-Tichadou and Fabrice Rastello. "Register Allocation". In: *SSA-based Compiler Design*. Ed. by Fabrice Rastello and Florent Bouchez-Tichadou. Springer, 2022, pp. 303–328. DOI: 10.1007/978-3-030-80515-9_22. URL: https://doi.org/10.1007/978-3-030-80515-9%5C_22.

[11] Aria Desires. *Introduction - Learning Rust With Entirely Too Many Linked Lists*. [Online; accessed 14. May 2025]. June 2024. URL: https://rust-unofficial.github.io/too-many-lists.

[12] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. "Stacked borrows: an aliasing model for Rust". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 41:1–41:32. DOI: 10.1145/3371109. URL: https://doi.org/10.1145/3371109.

[13] The Rust Project Developers. *The Rustonomicon*. [Online; accessed 28. Oct. 2024]. Oct. 2024. URL: https://doc.rust-lang.org/nomicon.

[14] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. "Points-to analysis using BDDs". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI '03. San Diego, California, USA: Association for Computing Machinery, 2003, pp. 103–114. ISBN: 1581136625. DOI: 10.1145/781131.781144. URL: https://doi.org/10.1145/781131.781144.

[15] Aaron Turon. *Fearless Concurrency with Rust | Rust Blog*. [Online; accessed 9. May 2025]. May 2025. URL: https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.

[16] Steve Klabnik, Carol Nichols, and Chris Krycho. *The rust programming language: 2nd edition*. en. Available online: https://doc.rust-lang.org/stable/book/.; With contributions from the Rust community. San Francisco, CA: No Starch Press, Feb. 2023. URL: https://nostarch.com/rust-programming-language-2nd-edition.

[17] Herb Sutter. *C++ safety, in context*. [Online; accessed 4. Oct. 2024]. Mar. 2024. URL: https://herbsutter.com/2024/03/11/safety-in-context.