Safety-Critical Software Development in C++

Daniel Kästner(✉), Christoph Cullmann, Gernot Gebhard, Sebastian Hahn, Thomas Karos, Laurent Mauborgne, Stephan Wilhelm, and Christian Ferdinand

AbsInt GmbH., Science Park 1, 66123 Saarbrücken, Germany
kaestner@absint.com

Abstract. The choice of the programming language is a fundamental decision to be made when defining a safety-oriented software development process. It has significant impact on code quality and performance, but also on the achievable level of safety, the development and verification effort, and on the cost of tool qualification. Traditionally, safety-critical systems have been programmed in C or ADA. In recent years, also C++ has entered into the discussion. C++ enables elegant programming, but its inherent language complexity is much higher compared to C. This has implications for testability, structural coverage, performance, and code analysis. Further issues to be considered are tool chain diversity, the role of the standard library, and tool qualification for compilers, analyzers and other development tools. This article summarizes the requirements of different safety norms, illustrates development and verification challenges and addresses tool qualification.

1 Introduction

During the past years the size and complexity of embedded software has sharply increased, in particular in the automotive domain. Contributing factors have been the trend to higher levels of automation, cost reduction by shifting functionality from hardware to software, and generic interfaces imposed by standardization frameworks such as AUTOSAR (AUTomotive OpenSystem ARchitecture) or Adaptive AUTOSAR.

A significant part of embedded software deals with safety-relevant functionality. A failure of a safety-critical system may cause high costs or even endanger human beings. Furthermore, due to increasing connectivity requirements (cloud-based services, device-to-device communication, over-the-air updates, etc.), more and more security issues are arising in safety-critical software as well.

Traditionally, safety-critical software is written in C or Ada. In recent years, also C++ is being in discussion, mainly because of its object-oriented language and abstraction features and its compatibility with C. Object-oriented programming can enable well-encapsulated programming, but also has drawbacks, as, e.g., a very implicit control flow structure. In the case of C++ there are also

A. Casimiro et al. (Eds.): SAFECOMP 2020 Workshops, LNCS 12235, pp. 98–110, 2020.
https://doi.org/10.1007/978-3-030-55583-2_7

other language properties which can make its use in safety applications hazardous unless the language is reduced to a safety-compatible subset and great care is taken.

Obviously, the safety of a system must be established whichever programming language is used, so safety-relevant C++ programs must satisfy the same safety and quality requirements as C or ADA programs. The selection of an appropriate programming language is a fundamental part of the definition of the software development process, and is one aspect in the definition of compliant software development life cycles and processes imposed by contemporary safety norms, including DO-178C, IEC-61508, ISO-26262, or EN-50128. That also includes the definition of appropriate coding guidelines to ensure safety-compliant use of a programming language. In addition, all relevant safety standards also require to identify potential hazards and to demonstrate that the software does not violate the relevant safety goals. The process of demonstrating that the requirements are satisfied is typically termed verification. Software tools used in such a safety process have to be appropriately qualified.

As of 2020, the discussion which C++ language features should be admitted to which extent for safety-critical software projects, is in full swing, and there is no clear consensus yet. This article attempts to give an overview: we address considerations to be taken into account when writing safety-critical C++ programs and discuss the implications of the programming language design on development, verification, analyzability, and tool qualification.

2 The C++ Language

In 1979, Bjarne Stroustrup started his work on extending the procedural programming language C with object-oriented features. Nowadays, after several revisions and extensions, C++ is a multi-paradigm language that supports not only procedural and object-oriented, but also generic as well as functional programming. Similar to C, C++ still supports low-level system programming but also appeals to high-level application programmers by offering a high degree of abstraction. The C++ language has been standardized first in 1998 by the International Organization for Standardization (ISO). Since then, it encountered revisions in 2011, 2014, and 2017, and the next revision C++2020 has been announced.

The C++ *standard library* covers useful infrastructure that is implemented in but not part of the core language itself. The language support library includes basic support for the run-time type information, dynamic memory allocation, and exception handling. The string library defines support for creating and manipulating sequences of characters while the input/output library provides utilities for communication with the outside world. The standard library defines a variety of containers to hold and manipulate data, e.g., vectors and arrays, including a standardized way of iterating these containers. The algorithm library maintains a collection of useful generic algorithm, e.g., to process containers. In later C++ versions, the standard library was significantly extended, e.g., by introducing unordered hash containers, a library for handling and using regular expressions, smart pointers, and support for multi-threading as well as atomic operations on shared data, parallel versions of many algorithm in the library, new types such as `optional` and `variant`, etc.

3 A Safety Standard's Perspective

The supplement DO-332 ("Object-Oriented Technology and Related Techniques") to DO-178C refines and extends the DO-178C objectives for usage of object-oriented technology. In particular, it lists language features of object-oriented programming that need to be taken into account when considering software safety, such as inheritance, parametric polymorphism, overloading, type conversion, exception management, and dynamic memory management. All these concepts are part of the C++ language, so these considerations immediately apply to safety-critical software development in C++. The DO-332 gives a definition of the relevant features, discusses their safety vulnerabilities and formulates dedicated verification objectives. It also lists generic issues for safety which always have to be taken into account and which might be negatively impacted by object-oriented language features. In the following we will summarize the overview of safety-relevant language features of the DO-332.

Inheritance. Classes can be considered as user-defined types, a subclass derived from a superclass then is a subtype. An important rule is that any subtype of a given type should be usable wherever the given type is required, otherwise the type system becomes unsafe. This is formally defined by the Liskov Substitution Principle (LSP): "Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be true for objects y of type S where S is a subtype of T". In particular that means that for all methods of a subclass which overload a superclass method the following requirements must be satisfied: preconditions may not be strengthened, postconditions may not be weakened, and invariants may not be weakened. The DO-332 requires the LSP to be demonstrated.

A derived class inherits properties from its parent class. In case the subclass may have several parent classes this is called *multiple inheritance*, otherwise *single inheritance*. With *interface inheritance*, the derived class shares only the complete signatures of the methods of its parents without the underlying implementation. With implementation inheritance, the inheriting class shares the implementation as well as the complete method signatures of the parent.

Method dispatch can be done statically based on the declared type of the object, or it can be done dynamically based on the actual type of the object. In C++ static dispatch happens on procedure calls, or when prefixing the invocation of a method m with a class type (`base_class::m()`). When a subtype of the static type redefines the method being called, the method of the static type is still called. When a method call is dynamic dispatched, the mapping of a specific implementation is performed at runtime. Each method is associated with an offset in the method table for the class of the object, so dispatching is an indirect procedure call. Vulnerabilities may be introduced by:

- Substitutability: When a method is overridden in a subclass such that it is functionally incompatible with the original definition, the LSP is violated and class substitution can cause an application to behave incorrectly.
- Method implementation inheritance: When a subclass has an additional attribute and a method which would need to update that attribute is not overwritten, a call to that method could result in an improper object state and unexpected behavior.
- Unused code: When a method of a superclass is overridden by the subclass and the superclass is not instantiated, the code will be unreachable; on the other hand, removing the method at the superclass may break its integrity.
- Dispatching: When static dispatch is used with method overriding, i.e., when the subclass overrides method `m`, the actual type is the subclass, and the method implementation of the superclass is invoked (`base_class::m()`), incorrect behavior, or state inconsistency might result.
- Multiple inheritance: Inheriting from two different classes with incompatible intentions may lead to unexpected behavior.

Polymorphism. Parametric polymorphism is a feature where a function takes one or several types as arguments treating the types uniformly, i.e., the function does the same kind of work independently of the type of its arguments. In C++ parametric polymorphism is realized via templates. *Ad-hoc polymorphism* means that the same function name (symbol) can be used for functions that have different implementations. In C++ ad-hoc polymorphism exists in the form of method overloading and method overriding. *Overloading* (*static polymorphism*) means that multiple methods have the same name but different types of arguments. Calls to overloaded methods are resolved at compile-time based on the type of call arguments. *Overriding* (*dynamic polymorphism*) means redefining a virtual method in a subclass. Calls to overridden methods are resolved at run-time based on the dynamic type of the object on which the call is invoked.

The parametric polymorphic operations that act on the generic data (template) may not be consistent with the substituted data. Also, source code to object code traceability is more difficult. In C++, one template class will be expanded into a different versions of the code for each type used. Overloading ambiguity occurs when a compiler performs implicit type conversions on parameters in order to select an acceptable match. This implicit type conversion could lead to an unintended method being called.

Type Conversions. 015A *narrowing conversion* is structural change in which the destination data element cannot fully represent the source data. A *downcast* is a change of view from supertype to subtype. A type conversion is called *implicit* if it is an automatic type conversion performed by the compiler. Although languages and compiler typically have strict implicit conversion rules, a lack of familiarity with these rules, or ignorance of the conversion taking place are often the cause of coding problems. With a narrowing type conversion, data may be lost. A downcast may result in data corruption of the object itself or its neighbors in memory, incorrect behavior, or a run-time exception being thrown.

Exception Management. Exception handling, i.e., the ability to throw exceptions within a method and to handle exceptions in calling methods, is a common feature of object-oriented languages. Exception handling is used for conditions that deviate from the normal flow of execution. When an exception is raised the execution will transfer to the nearest enclosing handler. In case of *checked exceptions* the subprogram signature lists all exceptions that can be thrown by the subprogram, and there are compile-time checks that corresponding handlers exist. *Unchecked exceptions* are not part of the subprogram signature so there can be no assurance that they will be handled.

Potential vulnerabilities include that an exception may not be handled, that either no action or inappropriate actions may be performed when the exception is caught (handled), or that operations are unexpectedly interrupted by exceptions. Exception handling can cause unexpected delays.

Dynamic Memory Management. With dynamic memory management, objects can be created on-demand during run-time. The deletion of such objects after their use is either performed explicitly using a program statement or automatically by a runtime system. The most important techniques are memory pooling, stack- and heap-based memory management. Dynamic memory allocation, typically stack and heap usage, can lead to free memory exhaustion and improper behavior can lead to memory corruption. The DO-332 lists the following potential vulnerabilities:

- Ambiguous references: An allocator returns a reference to live memory, e.g., an object that is still reachable from program code, allowing the program to use this memory in an unintended manner.
- Fragmentation starvation: An allocation request can fail due to insufficient logically contiguous free memory available.
- Deallocation starvation: An allocation request can fail due to insufficient reclamation of unreferenced memory, or lost references.
- Heap memory exhaustion: The size of the available memory could be exceeded.
- Premature deallocation: A memory fragment could be reclaimed while a live reference exists.
- Lost update and stale reference: In a system where objects are moved to avoid fragmentation of memory, e.g., an old copy might be accessed after the new copy has been created and is in use.
- Time-bound allocation or deallocation: Dynamic memory management could cause unexpected delays.

In particular for heap-based memory management, all vulnerabilities have to be addressed.

Structural Coverage. Structural coverage analysis provides evidence to which degree requirements-based testing exercised the code structure. It covers code coverage (statement coverage vs. decision coverage vs. MC/DC coverage), source code to object code traceability, control coupling and data coupling.

Object-oriented languages use control coupling to minimize data coupling by introducing control coupling via dynamic dispatching. Almost all object-oriented language features complicate determining the degree of structural coverage, inheritance, method overriding, dynamic dispatch, subprogram overriding with static dispatch and parametric polymorphism. In particular, code coverage itself does not take data and control coupling into account, e.g., covering a method in the context of the superclass or subclass alone may not exercise all possible execution conditions. To judge the extend of structural coverage reached by testing, sound static analysis can to be applied beforehand to determine class structure, inheritance hierarchy, and the maximal possible data and control coupling.

4 Coding Guidelines for C++

Many C software development projects are developed according to coding guidelines, such as MISRA C, SEI CERT C, or CWE (Common Weakness Enumeration), aiming at a programming style that improves clarity and reduces the risk of introducing bugs. In particular in safety-critical software projects, obeying suitable coding guidelines can be considered mandatory as this is strongly recommended by all current safety standards, including DO-178B/DO-178C, IEC 61508, ISO 26262, and EN 50128.

Safety norms do not enforce compliance to a particular coding guideline, but define properties to be checked by the coding standards applied. As an example, the ISO 26262 gives a list of topics to be covered by modelling and coding guidelines, including enforcement of low complexity, enforcing usage of a language subset, enforcing strong typing, and use of well-trusted design principles (cf. ISO 26262 2018, Part 6, Table 1). The language subset to be enforced should exclude, e.g., ambiguously defined language constructs, language constructs that could result in unhandled runtime errors, and language constructs known to be error-prone. Table 6 of ISO 26262:6 2018 lists some principles for software unit design that can be addressed by coding guidelines, such as having one entry and one exit point in functions, and avoiding dynamic objects/variables, multiple use of variable names, implicit type conversions, hidden data flow or control flow, and recursions. Similar requirements are imposed by IEC-61508 and DO-178C. In the following we list the most prominent C++ coding guidelines, and give a brief comparison focusing on safety-related requirements.

The MISRA C++:2008 coding guidelines [15] were released in 2008. The guidelines explicitly demand for a single point of function exit (Rule 6-6-5), ban any kind of heap memory allocation (Rule 18-4-1), require any variable to be initialized before use (Rule 8-5-1) and forbid recursion of any kind (Rule 7-5-4). Further rules impose restrictions on the use and uniqueness of entity names (including types, variables and scoping), and on the use of pointers, concerning among others (unsafe) pointer conversions, pointer arithmetic and the use of invalid pointers. Dangerous implicit type conversions are forbidden, e.g., downcasts (Rule 5-2-3) and implicit change of signedness (Rule 5-0-4). Similarly, the use of inheritance is further restricted, e.g. virtual inheritance shall be avoided in general (Rule 10-1-1) and only used if required in diamond hierarchies (Rule 10-1-2). Polymorphism in form of templates is regulated by another 10 rules addressing name lookup, instantiation and more. MISRA allows the use of exceptions, but restricts it to error handling (Rule 15-0-1) and gives further prerequisites that need to be fulfilled.

In 2018, the AUTOSAR consortium has released a set of guidelines for C++14 [1] aiming at complementing MISRA C++:2008 with respect to new language features. The guidelines cover new language features, but also relax some restrictions of MISRA C++:2008. In particular, dynamic memory allocation is allowed, though heavily restricted and bound to the use of C++'s memory management facilities (such as managed pointers). Exceptions can be used but their use is constrained by a set of 30 dedicated rules.

The Joint Strike Fighter Air Vehicle C++ Coding Standards [14] were released in 2005 and thus do not incorporate C++11 or later versions of the standard. A single point of exit per function is enforced (AV Rule 113), heap memory allocation is mostly forbidden (AV Rule 206) as well as recursion (AV Rule 119). The use of templates is restricted, e.g., tied to a dedicated code review (AV Rule 101) and exhaustive testing (AV Rule 102). The use of inheritance is also highly restricted by a set of 13 rules. The most notable difference with respect to the MISRA and AUTOSAR guidelines is that the use of exceptions is strictly forbidden (AV Rule 208), independently of the purpose.

Outside the domain of safety-critical systems many other coding guidelines have been proposed for C++. Most of them aim at reducing common weaknesses associated with certain language features and more or less strictly regulate their use. The use of exceptions is forbidden by the LLVM Coding Standards as well as the Google C++ Style Guide [5], while the SEI CERT C++ Coding Standard and C++ Core Guidelines [2] do not. Dynamic objects and resources are usually tied to the ownership concept and/or smart pointers. The Google C++ Style Guide also bans implicit conversions and restricts the use of inheritance, advocating the use of members/compositions instead where appropriate.

5 Analyzability

In this section we give an overview of static analysis as a standard verification technique recommended by all safety norms, and also summarize verification obligations that can be addressed by static analysis. However, the analyzability considerations described in the following sections also apply to other static verification techniques.

One application of static analysis is checking compliance to coding standards. Purely syntactical methods can be applied to check syntactical coding rules. Safety norms also require to demonstrate the absence of critical programming defects, such as runtime errors, stack overflows, or deadline violations. To find such defects – which to some degree can be covered by semantical coding rules –, semantics-based static analysis can be applied. Semantical analyzers can be further grouped into unsound vs. sound approaches, the essential difference being that in sound methods there are no false negatives, i.e., no defect will be missed (from the class of defects under consideration). Sound analyzers are based on a mathematically rigorous formal method for semantics-based static program analysis, called abstract interpretation [4].

Runtime errors due to undefined or unspecified behaviors of the programming language used are a particularly dangerous class of software errors. Examples are faulty pointer manipulations, numerical errors such as arithmetic overflows and division by zero, data races, and synchronization errors in concurrent software. Such errors can cause software crashes, invalidate separation mechanisms in mixed-criticality software, and are a frequent cause of errors in concurrent and multi-core applications. At the same time, these defects also constitute security vulnerabilities, and have been at the root of a multitude of cybersecurity attacks, in particular buffer overflows, dangling pointers, or race conditions [9].

In safety-critical C programs, the run-time stack (often just called "the stack") typically is the only dynamically allocated memory area. It is used during program execution to keep track of the currently active procedures and facilitate the evaluation of expressions. When the stack area is too small, a *stack overflow* occurs: memory cells from the stacks of different tasks or other memory areas are overwritten. This can cause crashes due to memory protection violations and can trigger arbitrary erroneous program behavior, if return addresses or other parts of the execution state are modified.

In real-time systems the overall correctness depends on the correct timing behavior: each real-time task has to finish before its deadline. Providing evidence that no deadlines are violated requires the worst-case execution time (WCET) of all real-time tasks to be determined.

Sound static analysis is often perceived as a technique for source code analysis at the programming language level. Run-time error analysis deals with unspecified and undefined behavior in the programming language semantics and therefore works at the source code level. However, sound static analysis can also be applied at the binary machine code level. In that case it does not compute an approximation of a programming language semantics, but an approximation of the semantics of the machine code of the microprocessor. Worst-case execution time analysis and worst-case stack usage analysis are performed at the binary level, because they have to take the instruction set and hardware architecture into account. In runtime error analysis, soundness means that the analyzer never omits to signal an error that can appear in some execution environment. In WCET and stack usage analysis soundness means that the computed WCET/stack bound holds for any possible program execution.

Nowadays, abstract interpretation-based static analyzers that can compute safe upper bounds on the maximal stack usage and on the worst-case execution time [8,10], and that can prove the absence of runtime errors and data races [11] are widely used for developing and verifying safety-critical software.

The complexity and precision of semantical analysis depends on the language semantics and the relevant language subset. For C++ code, the analyzability may be reduced due to certain language features discussed in the following.

6 C++ Challenges

All safety vulnerabilities presented in Sect. 3 are relevant for safety-critical C++ programs. In this section we revisit some of them and discuss additional topics, putting a particular focus on analyzability by static analysis tools. Differences in the impact to source- and binary-level analysis will also be discussed.

Rapid Language Evolution. Unlike in the older phases of the C++ language standardization, where more than a decade passed between two versions of the standard (e.g. C++98 to C++11), the evolution now happens at a much faster pace. The C++ standardization committee settled on a three-year cadence for new standard versions. This has led to the succession of the C++14, C++17 and soon C++20 standard. Each of the new versions of the standard incorporates both, new core language features and library extensions. Unlike other languages like C, for which new standard versions often only add minor changes, each of the C++ standard revisions is a rather large change. Compilers, analyzers and other development tools must keep up with the fast pace of this evolution.

Complex Language Frontend. Unlike the C language, the C++ language requires a highly complex frontend to support all current language features, e.g., to support template resolution. A C frontend is comparatively easy to implement and validate, and there is a plethora of different C frontends in use. In contrast, there is only a small set of frontends available that support modern C++, to our knowledge GCC/clang/MSCV/EDG. Qualifying such a frontend for safety-critical systems can be a challenge.

Compilation. Based on a formal executable semantics of C [12], the formally verified CompCert compiler has been developed. CompCert has been proven, using machine-assisted mathematical proofs, to be exempt from miscompilation issues: the executable code it produces is proved to behave exactly as specified by the semantics of the source C program [13]. The article [7] describes the qualification strategy used to successfully qualify CompCert for use in a highly critical control system from the nuclear power domain, compliant with IEC60880 and IEC61508 (SCL3). There is also a DO-178C-compliant Qualification Support Kit, that exhaustively maps the ISO C99 standard to functional tool requirements and, by a combination of formal proof and test cases, demonstrates 100% requirement coverage, which cannot be achieved for other existing compilers.

To the best of our knowledge no formal semantics for C++ has been proposed yet, which means that no comparable confidence in compiler correctness can be established.

The Standard Library. The C++ standard library provides abundant functionality. There is only a small number of library implementations available which cover the full functionality as required by the latest C++ standard.

From the perspective of the programmer, the high level of abstraction eases development and increases productivity. The underlying complexity is not apparent in the application code, but it can lead to negative effects for the analyzability of the resulting software. Many parts of currently available standard libraries hide dynamic memory allocation, creating the danger of complex allocation scenarios which are hard to analyze, and of unintentionally using dynamically allocated objects. The container parts of the standard library provide many conveniently usable data structures like associative maps (both ordered and unordered). Whereas the use is intuitive, the underlying data structures (highly dynamic pointer-based trees and hash tables) are sophisticated and make the analysis complex.

The commonly used standard libraries have not been developed according to safety standard requirements and are highly complex, e.g., the LLVM standard library consists of more than 800.000 lines of C++ code.

Library code is part of the safety-critical system and has either has to be developed with the same criticality level than the most critical component it is used in, or appropriately qualified (cf. DO-178C Sec. 12.1, ISO 26262:8).

Dynamic Memory Allocation. It is essential that for safety-critical systems adequate memory, processor and network resources are available to complete the tasks in a timely manner. Stack memory usage is well understood and can be efficiently and precisely handled by binary-level static analysis. With heap memory, the life range of an allocated object is not bound by the activity or the scope of the subprogram in which it was initially allocated. In consequence, its deallocation, in general, is not performed in the same context in which it was allocated, making its correct implementation a significantly greater problem than that of stack memory allocation and deallocation (DO-332, Sec. OO.D.2.4.2). It is necessary to determine the lifespan of each object, which is intractable, since exact lifespans depend on the data the program receives from its environment. Hence, manual allocation and reclamation of heap memory is error-prone, frequently leading to memory corruption through dangling pointers, and to memory leaks.

Dynamic heap memory allocation can also have impact on execution time, and sometimes also on the scheduling of time-critical tasks. The worst-case execution time often depends on the concrete addresses and their alignment on the target. For dynamically allocated data structures such properties are hard to derive. To the best of our knowledge, providing safe upper bounds for the time needed to allocate and free objects is an unsolved problem.

Even when the scope of dynamically allocated memory is well known, static analysis is challenging: the analysis problem becomes much harder to solve when the size of the state space cannot be precisely determined. The first difficulty is to keep the size of states representation finite (and small enough). In addition, because of approximations, the analyzer may consider more potential memory being allocated than is actually requested by the program during its executions, leading either to intractable memory consumption or further approximations on the allocated memories. Another aspect is that dynamic memory allocation makes using complex data structures, such as lists, trees or graphs, much easier for the programmer, which naturally tends to use them more. Such data structures are in themselves challenging to analyze, witnessed by the very active and abundant research on the subject of shape analysis [6].

Dynamic Polymorphism. The increased control coupling of dynamic dispatch means that the control flow is more dependent on the data flow in a program. In consequence, accurate information from data flow analysis has to be obtained to supplement control flow analysis (cf. DO-332, Sec. OO.D.2.4.1). The time needed to traverse method tables to find the correct method to invoke is relevant for WCET analysis.

The analysis is complicated by the need to determine at each dispatch point the set of methods inherited from a superclass or redefined by a subclass which might be invoked at that dispatch point. At the source level, even in case of non-perfect knowledge, the C++ type system will provide enough information to at least detect some super-set of the potentially called functions for function pointers and virtual functions. At the binary level however, identifying a virtual call site and its call targets needs additional information about the accessed object's base class type, and the called virtual member function including its signature. Determining this information typically requires debug information, and an explicit mapping from source code analysis to the binary level analysis, which, depending on the compiler optimization level, can be a challenge.

Exception Handling. C++ provides exceptions as one standard way for error handling, in addition to functions returning error codes. The C++ standard library use a mixture of exception handling and returning error codes. This renders uniform error handling difficult.

C++ exceptions are not checked by the compiler and there is no guarantee that an exception will be handled by the calling code. There is even no guarantee that throwing an exception succeeds: most C++ compilers generate code that indirectly invokes malloc in order to allocate heap memory for the exception object. If dynamic memory allocation fails, program execution is aborted by a call to `std::terminate`. An exception also may lead to unexpected program termination before reaching its handler, if the exception is raised in the scope of a destructor or in a method declared with the noexcept specifier.

Similarly to overloading, exception handling also massively increases the control coupling, as there is an additional implicit control path from every program point to all exception handlers in scope. For static analyzers this is particularly harmful, since analysis imprecision may cause additional over-approximations on calls to cleanup paths in case of an exception thrown. This, in turn, increases the complexity of the analysis and further reduces analysis precision.

At the binary code level, exceptions are hard to analyze at all, as the required stack unwinding is often handled via complex state machines that use extra information stored in debug sections (e.g., cf. [3]).

Extensions and/or modifications to the exception handling facilities of C++ are currently discussed in the C++ working group (WG21). More details can be found in [16], which proposes a model where functions have to declare that they throw a statically defined exception type by value, thereby making exception handling deterministic and avoiding dynamic or non-local overheads.

7 Summary

C++ has evolved to a complex multi-paradigm language which enables high programming productivity, in particular due to its powerful language abstractions. Object-oriented features like the C++ classes support well-structured programs, e.g., providing for encapsulation and reducing data coupling between components. However, for safety-critical programming the underlying concepts have to be well understood, since they typically have safety implications which have to be taken into account. Also, while supporting well-structured programs, the very same concepts may introduce safety defects when improperly used, cause hidden complexity in other aspects like an increased control coupling, or cause additional effort in other development stages, e.g., for achieving structural coverage. The expressiveness and complexity of the standard library constitutes a challenge for providing safety-compliant implementations, while rapid language evolution and frontend complexity represent challenges for tool chain qualification.

In this article en in DO-332, we have discussed language features which might have implications for software safety, in particular, inheritance, parametric polymorphism, overloading, type conversion, exception handling, and a higher temptation to use dynamic memory management. We also addressed C++ specific aspects, like its rapid language evolution, the central role of the standard library, and its frontend complexity. We have given an overview of the most important coding standards for C++, and shown that for some language features, in particular exceptions, there are significant differences in whether or to which extend their usage should be allowed. We discussed analyzability by static analysis tools, and showed that some language concepts can significantly increase analysis complexity and reduce the achievable confidence in the absence of defects.

References

1. AUTOSAR. Guidelines for the use of the C++14 language in critical and safety-related systems (2018)
2. Bjarne Stroustrup, H.S.: C++ Core Guidelines. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines. Accessed Jan 2020
3. C++ ABI for Itanium: Exception Handling. https://refspecs.linuxbase.org/abi-eh-1.21.html. Accessed Jan 2020
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4^{th} POPL, pp. 238–252. ACM Press, Los Angeles (1977)
5. Google C++ Style Guide. https://google.github.io/styleguide/cppguide.html. Accessed January 2020
6. Illous, H., Lemerre, M., Rival, X.: A relational shape abstract domain. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 212–229. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_15

7. Kästner, D., et al.: CompCert: practical experience on integrating and qualifying a formally verified optimizing compiler. In ERTS2: Embedded Real Time Software and Systems, Toulouse, France, p. 2018 (2018)
8. Kästner, D., Ferdinand, C.: Proving the absence of stack overflows. In: Bondavalli, A., Di Giandomenico, F. (eds.) SAFECOMP 2014. LNCS, vol. 8666, pp. 202–213. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10506-2_14
9. Kästner, D., Mauborgne, L., Ferdinand, C.: Detecting safety- and security-relevant programming defects by sound static analysis. In: Rainer Falk, J.-C.B., Chan, S. (eds.) The Second International Conference on Cyber-Technologies and Cyber-Systems (CYBER 2017), volume 2 of IARIA Conferences, pp. 26–31. IARIA XPS Press (2017)
10. Kästner, D., Pister, M., Gebhard, G., Schlickling, M., Ferdinand, C.: Confidence in timing. In: Safecomp 2013 Workshop: Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR), September 2013
11. Kästner, D., Schmidt, B., Schlund, M., Mauborgne, L., Wilhelm, S., Ferdinand, C.: Analyze this! sound static analysis for integration verification of large-scale automotive software. In: Proceedings of the SAE World Congress 2019 (SAE Technical Paper). SAE International (2019)
12. Krebbers, R., Leroy, X., Wiedijk, F.: Formal C semantics: CompCert and the C standard. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 543–548. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_36
13. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert - a formally verified optimizing compiler. In: ERTS: Embedded Real Time Software and Systems, 8th European Congress, Toulouse, France, p. 2016, January 2016
14. Martin, L.: Joint strike fighter air vehicle C++ coding standards for the system development and demonstration program (2005)
15. MISRA (Motor Industry Software Reliability Association) Working Group. MISRA C++:2008 Guidelines for the use of the C++ language in critical systems (2008)
16. Sutter, H.: Zero-overhead deterministic exceptions: throwing values. Technical report P0709 R0, SG14, May 2018