Doc. no. P2410r0
Date: 2021-07-12
Project: Programming Language C++
Audience: All
Reply to: Bjarne Stroustrup (bs@ms.com)

Type-and-resource safety in modern C++

Bjarne Stroustrup

1. Introduction

Complete type-and-resource safety have been an ideal (aim) of C++ from very early on (1979) and is achievable though a judicious programming technique enforced by language rules and static analysis. The basic model for achieving that can be found in [Str'15] and do not imply limitations of what can be expressed or run-time overheads compared to traditional C and C++ programming techniques. The basic design and enforcement techniques simply ensure that:

- §2: every object is accessed according to the type with which it was defined
- §3: every object is properly constructed and destroyed
- §4-5: every pointer either points to a valid object or is the **nullptr**
- §6: every reference through a pointer is not through the **nullptr** (often a run-time check)
- §6: every access through a subscripted pointer is in-range (often a run-time check)

That's just what C++ requires and what most programmers have tried to ensure since the dawn of time. The difficulty is to guarantee it in a realistically-sized program. Experience shows that this cannot be done without static analysis and run-time support. Furthermore, for fundamental reasons this cannot done even with such support if arbitrary legal language constructs are accepted while conventional good performance must be maintained.

The way out of this dilemma is a carefully crafted set of programming rules supported by library facilities and enforced by static analysis.

This presentation is based on the C++ Core Guidelines [CG] and their enforcement rules (e.g., as implemented by the Core Guidelines checker distributed with Microsoft Visual Studio). That is, the points made here are backed up by specific rules and supported by existing software. By default, the CG do not provide complete type-and-resource safety. This paper is a high-level overview of the rules that must be enforced to guarantee that. Details can be found elsewhere (§9).

All static analysis for GC is local; that is. Non-local static analysis, e.g., whole-program analysis, is not scalable and can't in general handle dynamic linking.

To meet common definitions of safety, we further need to address

- Narrowing conversions and overflow
- Data races and deadlocks

Those are dealt with separately in the CG and elsewhere.

The Core Guidelines are designed for selective and gradual adoption. Consequently, all traditional C++ technique are available where conversion to stricter rules is not considered practical. In particular, nothing must impede C++'s ability to directly manipulate hardware where and when necessary.

2. Object access

Every object is accessed according to the type with which it was defined.

The language guarantees this except for preventable pointer misuses (see §4, §5), explicit casts, and type punning using **union**s. The CG have specific rules to enforce these language rules.

Static analysis can prevent unsafe casting and unsafe uses of **union**s. Type-safe alternatives to **union**s, such as **std::variant**, are available. Casting is essential only for converting untyped data (bytes) into typed objects.

3. Construction and destruction

Every object is properly constructed and destroyed.

Static analysis easily prevents the creation of uninitialized objects. Buffers of uninitialized **unsigned char**s are acceptable according to the language definition and needed for performance reasons.

The language guarantees that destructors for scoped objects are invoked upon scope exit and that destructors for static objects are invoked upon program termination.

Using copy elision or move operations, objects can be safely moved between scopes. The CG insist that a moved-from object be assignable, but general operations are not allowed on moved-from objects. This is ensured through static analysis.

Entities that must be acquired and later released for some other part of a system (e.g., memory or file handles) are called *resources* and represented as objects with destructors doing the release and often with constructors that do the acquisition as part of establishing an invariant. This is often referred to as *resource safety* or as RAII (*Resource Acquisition Is Initialization*). In addition to resource safety, this scope-based resource management ensures predictability and minimizes resource retention.

4. No dangling pointers

Every pointer either points to an object or is the nullptr. The first and essential step to ensure this is to guarantee initialization (see §3).

In this paper

- "pointer" includes all ways of referring to an object, including containers of pointers, references, lambda captures, and smart pointers.

- "return" includes all ways of getting a pointer value out of a scope, including containers of pointers, reference arguments, pointers to pointers, lambda captures, global variables, and exception values.

4.1. Escaping pointers

No pointer may point to an object after the object has gone out of scope. This is achieved through static analysis, preventing a pointer from "escaping" into a scope surrounding the object to which it points. A pointer value can be returned from a scope provided

(1) It was passed into the scope (e.g., as an argument or retrieved from an object external to the scope).
(2) It points to an object external to the scope (e.g., it was initialized by **new**).

If static analysis cannot prove that, the pointer cannot be returned. This implies limitations to the complexity of the flow of control leading to the return of a pointer value.

4.2. Invalidation

No pointer may access a **deleted** object. This trivially prohibits access to a **delete**d object in the scope in which it was created using **new** (and scopes nested therein). Like for detecting escaping pointers, this implies limitations to the complexity of the flow of control leading to the return if a pointer value.

This leaves the problem of preventing a pointer to an object on the free store from being **delete**d in a called function and then accessed through in its original scope. In principle, that could be handled using static analysis, but global static analysis is impractical or unaffordable in many contexts, so the CG resort to annotation:

- A pointer returned by **new** is an **owner** and must be **delete**d (unless stored in **static** storage to ensure that it lives "forever.").
- Only a pointer known to be an **owner** can be deleted. Thus, a pointer passed into a scope as an **owner<T*>** must be **delete**d in that scope or passed along to another scope as an **owner**. A pointer that is passed into a scope as a plain **T*** may not be **delete**d.
- A pointer passed to another scope as an **owner** and not passed back as an **owner** is said to be invalidated and cannot be used again in its original scope (since it will have been **deleted**).

Anything that holds an **owner** is subject to the owner rules. Given **owner** annotation, these rules are enforced by static analysis.

The **owner** annotation is necessary only for low-level code implementing higher-level abstractions (such as **vector**) and for pointers in interfaces that cannot be changed (e.g., for ABI reasons). The CG recommend preferring higher-level abstractions, such as **vector** and **unique_ptr** and avoid explicit **owner** annotations wherever possible.

4.3. "Odd" pointers

It is necessary to avoid access through a one-past-the-end pointer (e.g., an iterator returned from **find()**) and to avoid all-but-assignments to moved-from objects. This is ensured through static analysis enforcing proper use. A **not_end** type similar to **not_null** may be useful to help the static analyzer in cases where the result of **x.find()** and the like isn't immediately tested against **x.end()**.

5. Memory pools

The discussion in §4 assumes that objects are static, local (automatic), or on the free store (heap, dynamic memory) managed by **new** and **delete**. However, user-defined memory management in various forms is essential in many application areas and fundamental in the C++ standard library.

By a *memory pool*, I mean a section of memory in which an object can be stored. In principle, a pointer to an object in a memory pool can be handled in a similar manner to that of a pointer to an object created by **new**. However, C++ lacks a standard "pool" abstraction. Instead, there are thousands of variations of the idea, seriously complicating the task of static-analyzer writers.

To avoid dangling pointers to its stored objects, a pool can apply one of alternative strategies:

1. Disallow objects to be **delete**d or relocated
2. Disallow pointers to objects to escape
3. Invalidate all pointers to objects if a potentially deleting or relocating operation is invoked

std::vector with subscripting and **resize()** is a typical example of a pool that requires special attention and is dealt with through invalidation (the third alternative) enforced by static analysis. If a non-**const** function is invoked on a **vector**, all pointers to its elements are considered invalid and may not be used. This is ensured through static analysis. This is a conservative, but safe, strategy that can be applied to every pool. To enable a non-**const** function (e.g **vector::operator[]()**) to be considered not invalidating, we might add a **[[not_invalidate]]** annotation. Such annotation can be validated by static analysis.

6. No Range errors

Every reference through a pointer is not through the nullptr (often a run-time check).

The CG simply prohibit access through a pointer that is not known to be not the **nullptr**. As an alternative to repeated **nullptr** checks, it offers the **gsl::not_null** type.

Every access to an array is in-range (often a run-time check).

The CG simply prohibit subscripting of pointers (and equivalent address arithmetic). As an alternative, it offers **gsl::span** that provides range-checked access (a version of **gsl::span** is now **std::span**). Containers, range-**for**, and algorithms dramatically reduces the need for subscripting pointers compared to C-style code. Spans are ideally used in interfaces, but can also be used locally as an alternative to direct use of pointers passed through potentially unsafe interfaces; such pointers typically require special (see §7) attention or run-time checking.

7. Low-level code

C++ is extensively used for low-level manipulation of memory and other system resources. Making C++ safe by eliminating all direct access to "raw" memory is not an option. Languages that ban such unsafe access, typically have ways of allowing unsafe code or delegate such manipulation to code written in C or C++.

The current solution for messy, low-level code (e.g., for a memory manager where casts and pointer manipulation are necessary or for highly-optimized implementation of key data structures) is to apply the static analysis selectively. We may need a notion of "trusted code" marked in the code itself, maybe

indicated by a **[[trusted]]** annotation. Such annotation would allow programmers to understand **[[trusted]]** code independently of static analyzer settings. Naturally **[[trusted]]** code would require significant extra care and review; it should be minimized. Calls to other code from **[[trusted]]** code would be assumed correct by the static analyzer.

Such annotation need not be all-or-nothing. The "profile" options currently used to control the CG static analysis would make a good initial set of options, e.g. **[[trusted lifetime]]** would suppress the checking for leaks, etc.

For most code written in a modern C++ style, conforming to the restrictions needed to achieve type-and-resource safety doesn't require major structural changes or imply run-time overheads. Older styles of code will need to replace use of arrays through pointers with abstractions such as **vector** and **span**. However, there are structures that cannot easily be fitted into this guaranteed framework. An example is general graphs with nodes where ownership and lifetime aren't clearly indicated so that type-and-resource safety depends on the cleverness of the programmer. One possible solution is to cleanly and explicitly separate ownership and access (e.g., a **vector** of **owner** pointers plus a data structure on non-owning link pointers). Another is to use of a smart pointers (e.g. **std::shared_ptr**) plus tests for circularities.

8. So what?

The static analysis I rely on for guarantees is not yet 100% implemented (but is getting close) and what is available is not available on every platform. It would be a massive advantage for all C++ developers if it were. Universal availability of Core Guidelines static analysis would be far more significant than any single language extensions, and far easier/cheaper to achieve. It would also be following the tradition of C and C++ in distinguishing between what is legal in the standard and what is good software development. The compiler is not our only tool, and has never been.

What is checked statically should be principled and precisely specified. The C++ Core Guidelines is a significant effort in that direction. Also, the completeness of the safety guarantees needs to be – as far as possible – formally proved.

9. References

- [CG] The C++ Core Guidelines.
- The Core Guidelines Support Library (GSL).
- T. Ramananandro, G. Dos Reis, and X. Leroy: A mechanized semantics for C++ object construction and destruction, with applications to resource management. ACM/SIGPLAN Notices 2012/01/18.
- [Str'15] B. Stroustrup, H. Sutter, and G. Dos Reis: A brief introduction to C++'s model for type- and resource-safety. Isocpp.org. October 2015. Revised December 2015.
- B. Stroustrup: C++ -- an Invisible Foundation of Everything. ACCU Overload No 161. Feb'21.
- B. Stroustrup: Thriving in a crowded and changing world: C++ 2006-2020. ACM/SIGPLAN History of Programming Languages conference, HOPL-IV. London. June 2020.
- H. Sutter: Lifetime safety: Preventing common dangling. P1179R1. 2019-11-22.
- A Microsoft guide to using the Core Guidelines static analyzer in Visual Studio.