Title:
[WIP] Enriching C++ with Intuitive Container Queries: A Proposal for std::contains and Its Variants

Date:
2024-02-08

Author:
Robert L. Sitton, Jr.

Project:
ISO/IEC 14882: Programming Language C++

Abstract:
The introduction of std::contains utility variants in the C++ language would improve code readability, efficiency, and safety in handling common tasks involving containers and ranges. These functions would enhance code readability and expressiveness, reducing the need for boilerplate code and increasing the chance of errors. They would also improve efficiency by allowing optimizations specific to certain container types, such as containers with direct access or those maintaining a sorted order. This would also reduce type safety and error reduction by providing a standardized way to perform these checks, ensuring that comparisons are made between compatible types. Additionally, these functions would align with other languages, lowering the barrier to entry for developers coming from other languages and aligning C++ with common programming paradigms. The inclusion of these functions would also support modern C++ practices, enhancing the language's utility and developer experience.

1. Introduction

In the spirit of continuous improvement and recognition of the common patterns emerging in modern C++, the introduction of the following functions is proposed: std::contains, std::contains_any, and std::contains_all.

2. Motivation and Scope

The current standard needs a more direct, standardized way to query containers for the presence of elements, requiring developers to either write verbose code or rely on less intuitive workarounds. This gap hinders code readability and maintainability and deviates from the C++ philosophy of providing robust abstractions that reflect and refine how developers think and work.

3. Rationale

std::contains: Offers a straightforward way to check for the presence of a single element.

std::contains_any and std::contains_all: Extend this querying capability to sets of elements, enabling expressive and efficient checks that align with everyday programming needs.

These functions, by their design, promote code that is easier to write and understand, reflecting the philosophical commitment of C++ to serve as a bridge between the abstract elegance of theory and the concrete demands of practice.

4. Impact on Existing Code

Introducing these functions will be entirely additive, enhancing the standard library without breaking existing code. Their design encourages optimal implementation, offering performance and stability benefits over ad-hoc solutions.

5. Technical Specifications

The proposed utility functions support various container types, including sequences and associative containers. By leveraging concepts and SFINAE, the functions can provide compile-time checks that guide correct usage, embodying the C++ philosophy of catching errors early.

6. Implementation

Preliminary implementations demonstrate the feasibility and benefits of these functions. Performance benchmarks and use cases are provided to illustrate their efficiency and utility across various scenarios.

```
// Check if a container contains a specific value
template <typename Container, typename Value>
bool contains(const Container& container, const Value& value) {
  return std::find(std::begin(container), std::end(container), value) != std::end(container);
}

// Check if a container contains any of the specified values
template <typename Container, typename... Values>
bool contains_any(const Container& container, Values&&... values) {
  return (... || contains(container, std::forward<Values>(values)));
}

// Check if a container contains all of the specified values
template <typename Container, typename... Values>
bool contains_all(const Container& container, Values&&... values) {
  return (... && contains(container, std::forward<Values>(values)));
}

// Check if a range contains a specific value
template <std::ranges::input_range Range, typename Value>
bool contains(const Range& range, const Value& value) {
  return std::ranges::find(range, value) != std::ranges::end(range);
}
```

```
// Check if a range contains any of the specified values
template <std::ranges::input_range Range, typename... Values>
bool contains_any(const Range& range, Values&&... values) {
  return (... || contains(range, std::forward<Values>(values)));
}

// Check if a range contains all of the specified values
template <std::ranges::input_range Range, typename... Values>
bool contains_all(const Range& range, Values&&... values) {
  return (... && contains(range, std::forward<Values>(values)));
}
```

7. Proposed Wording

The detailed wording for the standard will specify the exact signatures, constraints, and behaviors of std::contains, std::contains_any, and std::contains_all, ensuring they integrate seamlessly with existing container and algorithm facilities.

8. Conclusion

By introducing std::contains and its variants, C++ takes a significant step toward a more intuitive and expressive standard library.

References

Rainer Grimm, "The Ranges Library in C++20: Design Choices," MC++ BLOG, October 30, 2023. This reference discusses the introduction of the ranges library in C++20, highlighting the design choices to enhance working with the Standard Template Library (STL) through features like laziness, direct operation on containers, and composition. It underscores the evolution of C++ towards more expressive and efficient code, supporting the rationale for further enriching the language with container query utilities.

Rainer Grimm, "C++23: Four new Associative Containers," MC++ BLOG, September 4, 2023. This article introduces std::flat_map, std::flat_multimap, std::flat_set, and std::flat_multiset in C++23 as replacements for their ordered counterparts, motivated by improvements in memory consumption and performance. The evolution of container types in C++23 underscores the ongoing efforts to optimize data structures for efficiency, which aligns with the proposal's aim to introduce utility functions that can leverage these optimizations for better performance and usability in container queries.

Related work

Related work in other programming languages that have influenced or could provide context for proposing std::contains, std::contains_any, and std::contains_all in C++ includes:

These examples from other programming languages showcase a trend toward providing developers with more intuitive, expressive, and concise ways to query collections. By

incorporating similar functionalities into C++, the proposal aims to align C++ more closely with modern programming practices, enhancing its utility and ease of use in a way consistent with trends in software development across various languages.

Python:
Python's list, set, and dictionary types have direct methods for checking containment using the in a keyword, such as if an element is in the list, which simplifies querying collections. The any() and all() functions in Python allow for expressive, concise checks over iterables, aligning closely with the proposed std::contains_any and std::contains_all.

Java:
Java's Collection Framework includes methods like contains(Object o) in the Collection interface, providing a unified way to check for an element's presence across different collections. Java Streams introduced in Java 8 further enable complex queries with methods like anyMatch(), allMatch(), and noneMatch(), offering similar functionality to what is proposed for C++.

Rust:
Rust provides methods like contains(&self, value: &T) for data structures such as Vec and HashSet, facilitating element containment checks. Additionally, Rust's iterators offer methods like any() and all() for predicates over iterables, showcasing a modern approach to collection querying that emphasizes safety and expressiveness.

Swift:
Swift's Collection Protocol includes the contains(_:) method for checking if a collection contains a specified element. Like Rust and Python, Swift also offers contains(where:) and high-order functions like allSatisfy(_:) on sequences for more complex queries, emphasizing expressive and functional programming patterns.

JavaScript (ECMAScript):
JavaScript arrays have methods like includes(element) for containment checks and functions like some() and every() for testing predicates against all elements of an array. These functions reflect the language's flexible approach to working with collections in a high-level, interpreted context.

Examples:

```
#include <iostream>
#include <vector>
#include <algorithm>


int main() {
    std::vector<int> const activeDetectors = {101, 102, 103, 104};
    std::vector<int> const requiredDetectors = {301, 302, 303};
    int const detectorToCheck = 102;


    if (contains(activeDetectors, detectorToCheck)) {
        std::cout << "Detector " << detectorToCheck << " is currently active."
<< std::endl;
    } else {
        std::cout << "Detector " << detectorToCheck << " is not active." <<
std::endl;
    }


    if (contains_any(activeDetectors, 104, 205, 206)) {
        std::cout << "At least one of the specified detectors is active." <<
std::endl;
    } else {
        std::cout << "None of the specified detectors are active." <<
std::endl;
    }


    if (contains_all(requiredDetectors, 301, 302, 303)) {
        std::cout << "All required detectors are active for the experiment." <<
std::endl;
    } else {
        std::cout << "Required detectors are not active for the experiment." <<
std::endl;
    }
}
```