Doc. No. P3406R0
Date: 2024-09-22
Audience: All
Reply to: Bjarne Stroustrup (bjarne@stroustrup.com)

We need better performance testing

Bjarne Stroustrup

Abstract

Does the C++ design follow the zero-overhead principle? Should it? I think it should, even if that principle isn't trivial to define precisely. Some of you (for some definition of "you") seem not to. We – WG21 as an organization – haven't taken it seriously enough to make it a requirement for acceptance of new features. I think that is a serious problem, but one that we (WG21) should be able to handle. This paper offers some examples.

1. Introduction

What is the "zero-overhead principle"? The simplest articulation can be found in D&E:

- What you don't use, you don't pay for.

This specifically aims to eliminate "distributed fat" such as helpful information added to every object (e.g., type identification to a complex number) or redundant tests. Such helpful information or tests can be requested when and where they are desired.

The principle was conceived as applying to run time only. Issues such as compile time and integration with existing tools can and should be considered separately.

"Zero-overhead" should always be considered relative to requirements. It is not an absolute requirement that can be brainlessly applied. By simplifying a set of requirements (e.g., "works only for trivial types") you can almost always gain some performance, though for templates, specialization often allows us to match such simplifications. The D&E rules specifically suggest specialized implementations (e.g., machine code) for important special cases.

The zero-overhead principle was/is meant to apply to both language features and library features. For library features, we often add something like:

- What you do use, you can't write more efficiently using lower-level C++ features.

With very few exceptions, C++ followed the Zero-overhead rule when I wrote it (the language and the principle). Does it still? To what extent should it do so in the future?

My conclusion is that every new language feature and standard-library component proposal should contain a discussion of its relation to the zero-overhead principle.

2. Examples

Here, I give a few examples of design decisions that might have been improved and/or better understood by more developers, had a discussion of zero-overhead been prominently documented at the time.

2.1. Unsigned indices

A prominent argument for using unsigned indices for the standard-library containers was that doing so saved a test when range checking. For example:

```
void f(int i, unsigned ui)
{
        If (v.size()<=ui) error();          // unsigned style (test against one value)
        If (i<0 || v.size()<=i) error();    // signed style (test against two values)
}
```

Was the performance difference significant? The lower value tested against is essentially always the constant zero. Is it significant today?

In this case, I don't think the zero-overhead argument was a valid reason for using unsigned. Examining the alternatives carefully and documenting the choice might have saved millions of programmers from troubles (and memory-safety problems) stemming from mixing signed and unsigned values.

2.2. Range-for

I am a great fan of range-**for**. It is terser that a C-style **for**-loop and offers fewer opportunities for errors. But is it always as performant? Consider

```
for (auto x : v) do_something();
for (auto& x : v) do_something();
for (auto&& x : v) do_something();
for (int i= 0; i<v.size(); ++i) do_something();
```

Are the performances always equivalent? Why not? When do the differences matter? What are appropriate rules of thumb for choosing among the alternatives. A discussion of this would have improved the proposal at the time and helped innumerable users.

2.3. RTTI

Run-time type information was required only for classes that had at least one virtual function. That way, the RTTI could be (logically) attached to the virtual function table and have no impact on object sizes. For some, that seemed too much overhead and implementations acquired "no RTTI" options and in places the use of dynamic_cast became demonized with the result that we suffered bugs from misuses of static casts.

At the time, I estimated the memory overhead for RTTI to be insignificant and worthwhile for the utility and protection against class hierarchy navigation errors it offered. Had I documented my

argument and size estimates thoroughly, we may have been in a better situation today. Had I been wrong about the overhead, we might have discovered that and also been in a better situation.

2.4. Range checking

For raw performance, range checking of every subscript was unaffordable. In many applications, it still is. However, range errors are a significant problem and I argue for optional range checking for all subscripting (Profiles https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3274r0.pdf). At the time (1985 or so) I was hoping for hardware support for range checking. I am still waiting for the research in that area to make it into the mainstream. The use of spans, algorithms, and range-**for** can dramatically reduce the cost of range checking. I suspect that some organizations have data that could help rational decision making. Such as, where is range-checking overheads significant and by how much?

2.5. Exceptions

At the time, I argued that exceptions obeyed the zero-overhead principle. I think I was correct in that, but only when exceptions are used as intended and I wasn't successful at explaining that intended use. Exceptions were never intended to be used for every "error" or as simply an alternative way of returning a value. I pointed that out at the time but failed to get the idea through to significant user groups.

Two implementations were tried at the time:

- Returning and implicitly testing an exception value (a bit like systematic use of optional or expected).
- Keeping the ordinary return path conventional and efficient while relying on a lookup table for the exceptional path.

The second alternative won out, reflecting the view that

- errors that are expected (e.g., failure to open a file) should be handled explicitly and locally.
- only rare errors (e.g., network separation) could be considered exceptional.
- error handlers are rare compared to function definitions.

These discussions were not gathered in one place and updated based on experience. That did major harm. There is at least a chance that a specific and well-known discussion of cost (zero-overhead) as a standard part of the documentation of the design would have focused on the discussion of proper use.

Exceptions can lead to simpler code than explicit error code and even be significantly more efficient than use of error code. Recently, this has been documented even for micro controllers (Khalil Estell: C++ Exceptions for Smaller Firmware). The same work demonstrated that current exception handling support is seriously sub-optional. Arguments, such as I have made them "forever" in books and presentations don't have the effect of measurements.

2.6. Expected

I have seen **std::expected** presented as a more principled and more efficient alternative to using exceptions. Here, I will address only the efficiency argument. Consider this simplified variant of an example from cppreference.com:

```
auto parse_number(std::string_view& str) -> std::expected<double, parse_error>;

void process(std::string_view str)
{
        if (const auto num = parse_number(str); num.has_value())
                std::cout << "value: " << *num << '\n';
        else
                handle_error();
};

for (auto src : {"42", "42abc", "meow", "inf"})
        process(src);
```

The rough equivalent using exceptions directly would be

```
auto parse_number(std::string_view& str) -> double;

void process(std::string_view str)
{
        std::cout << "value: " << parse_number(str)  << '\n';
};

for (auto src : {"42", "42abc", "meow", "inf"})
        process(src);
```

This assumes that formatting errors are rare and handled somewhere up the call chain. If that's not the case for an example, the examples are not equivalent and exceptions should not be used (iostreams supports both alternatives). Discussions of when using exceptions is appropriate can be found elsewhere (e.g., "A Tour of C++ (3rd edition)" §4.4). However, the obvious difference here is that the return value for the expected version is two values (rather than one) and an additional (and explicit) test (rather than none) is required. If this style is widely used, it implies "distributed fat" (in terms of the original conception of the zero-overhead principle) as well as verbosity.

When exceptions are not used for error propagation, repeated checks and repeated added error-code transmission can add significant "fat."

Incidentally, use of **std::expected** implies use of exceptions and potential for UB. For example:

```
for (auto src : {"42", "42abc", "meow", "inf"})          // "meow" is not a number
        cout << *src;   // UB

for (auto src : {"42", "42abc", "meow", "inf"})          // "meow" is not a number
```

```
cout << src.value();     // throws std::bad_expected_access
```

I would very much have liked an extensive discussion of performance issues related to the use of **std::expected** compared to conventional use of exceptions, conventional use of error-codes, an abstraction that eliminated the opportunity for UB, and an abstraction that eliminated the possibility of ignoring an error. Not all of the questions would be part of a discussion of the zero-overhead principle applied to **std::expected**, but much would have been exposed by a discussion and documentation of performance and likely common uses. We would not have had to guess about the intent of the proposal.

2.7. Pipes and views

Pipes are important in many areas and views are often a splendid way of passing information around. However, I worry about the complexity of the code implementing pipes and views in the standard library. I haven't found a discussion of the performance issues anywhere. Like many (most) people, I have not found the time to do a thorough analysis myself and I find it very hard to recommend the facilities when I don't know when and where to expect overheads. Thus, the lack of documented analysis of the facilities relative to the zero-overhead principle leads to disuse as well as potential reputational damage to C++ from overheads being detected late.

Also, has anyone compared the performance of the standard-library pipes with coroutine-based pipelining?

3. The standard counter argument

Since the birth of C++, I have regularly been told that efficiency doesn't matter anymore because the hardware is – or soon will be – fast enough to erase any language-related efficiency problems. That's indeed the case for some uses and users.

However, for all of its life, C++'s strength has been that it can do hard tasks efficiently and with some elegance. This must not be lost. And if the ultimate efficiency cannot be obtained through "usual means" there is always the option of tuning using intrinsics, inline assembly, or specialized hardware accessed through the language.

4. Suggestion

Every proposal, language and library, should be accompanied by a written discussion, ideally backed by measurements for credibility, to demonstrate that in the likely most common usage will not impose overheads compared to current and likely alternatives. Also, to demonstrate that optimizations and tuning will not be inhibited. If no numbers are available or possible, the committee should be extremely suspicious and in particular, never just assume that optimizations will emerge over time.

This should be a formal requirement.

Yes, that could be seen as rudely doubting proposers, but as a committee it is (also) our job to guard against our own overenthusiasm and group-think. Proposers should understand that.

How the zero-overhead principle relates to a feature is not just about performance. It cannot be discussed adequately without articulating what are likely common use cases and how they are best represented in code. This is a key design point in its own right.

A request for implementation experience is not the same as a request for a zero-overhead discussion.