that even if the virtual `deallocate` function were inline, unless it is declared **`final`** or the runtime type is somehow known at compile time, there is no sure way for the compiler to know that the function isn't overridden by a derived type.

In the case of `TypeB`, however, the function is both *declared* **`final`** and *defined* **`inline`**; hence, the virtual dispatch can be reliably sidestepped, the empty function can be inlined, and a true no-op is achieved with no runtime overhead.

Potential Pitfalls

Contextual keywords are contextual

Historically, the Standards Committee has taken different approaches to adding new keywords to the language. C++11 added ten new keywords to the language — **`alignas`**, **`alignof`**, **`char16_t`**, **`char32_t`**, **`constexpr`**, **`decltype`**, **`noexcept`**, **`nullptr`**, **`static_assert`**, and **`thread_local`**[7] — and thus made ten potential tokens no longer usable as identifiers. When considering new keywords, much effort is expended to determine the impact of that word's change in status on existing codebases. Two identifiers, **`override`** and **`final`**, were not made keywords and were instead given special meaning when used in contexts where previously identifiers were not syntactically allowed. This approach avoided possible code breakage for any existing codebases using these words as identifiers, at the cost of occasional confusion.

When used after a function declaration, **`override`** and **`final`** do not add any significant parsing ambiguity to the language; arbitrary identifiers were not syntactically valid in that position anyway, so confusion is minimal. When used on a **`class`** declaration, however, `final`'s meaning is not determined until tokens after it are parsed to distinguish between a variable declaration and a class definition:

```
struct S1 final;      // Error, variable named final of incomplete type
struct S2 final { };  // OK, final class definition
struct S2 final;      // OK, variable named final of complete type S2
```

Notice that the variable declarations in the example above both look like they might be an attempt to forward-declare a **`struct`** that is final but are instead a totally different language construct.

Systemic lost opportunities for reuse

Both **`final`** and **`override`** are similar in their complexity yet different in the potential adverse implications that widespread use can impose. Such ubiquitous use will depend heavily on the scale and nature of the development process employed. In some development

[7] C++14 and C++17 added no new keywords. C++20 added **`char8_t`**, **`co_await`**, **`co_return`**, **`co_yield`**, **`concept`**, **`consteval`**, **`constinit`**, and **`requires`**, notably mixing some words potentially already used as identifiers (**`concept`** and **`requires`**) with a collection of more obscure words that had little chance of conflicting with existing codebases.