

# Performance & Memory Considerations

Let's have a quick look at the performance and memory considerations of `std::variant`.

## WE'LL COVER THE FOLLOWING



- Migration From `boost::variant`

`std::variant` uses the memory in a similar way to union: so it will take the max size of the underlying types. But since we need something that will know what the currently active alternative is, then we need to use some more space. Plus everything needs to honour the alignment rules.

Here are some basic sizes:

```
#include <iostream>
#include <variant>
#include <string>

int main() {
    std::cout << "sizeof string: " << sizeof(std::string) << '\n';

    std::cout << "sizeof variant<int, string>: " << sizeof(std::variant<int, std::string>) << '\n';
    std::cout << "sizeof variant<int, float>: " << sizeof(std::variant<int, float>) << '\n';
    std::cout << "sizeof variant<int, double>: " << sizeof(std::variant<int, double>) << '\n';
}
```



On GCC 8.1, 32 bit:

```
sizeof string: 32
sizeof variant<int, string>: 40
sizeof variant<int, float>: 8
sizeof variant<int, double>: 16
```

What's more interesting is that `std::variant` won't allocate **any extra space!**  
No dynamic allocation happens to hold variants or the discriminator.

To have a safe sum type, you pay with an increased memory footprint. The additional bits might influence CPU caches. That's why you might want to do some benchmarking for the hot spots in your application that uses variants.

## Migration From `boost::variant` #

Boost Variant was introduced around the year 2004, so it was 13 years of experience before `std::variant` was added into the Standard. The STL type draws from the experience of the boost version and improves it.

Here are the main changes:

Feature	Boost.Variant (1.67.0)	<code>std::variant</code>
Extra memory allocation	Possible on assignment, see <a href="#">Design Overview - Never Empty</a>	No
visiting	<code>apply_visitor</code>	<code>std::visit</code>
get by index	no	yes
recursive variant	yes, see <a href="#">make_recursive_variant</a>	no
duplicated entries	no	yes
empty alternative	<code>boost::blank</code>	<code>std::monostate</code>

---

In the next lesson, we discuss the idea of `ErrorCode` in `std::variant`.

