# Performance & Memory Considerations

This module provides a comparison of std::any with std::variant and std::optional evaluating based on efficiency and memory allocations.

`std::any` looks quite powerful, and you might use it to hold variables of variable types... but you might ask what the price is for such flexibility.

## The Main Issue: Extra Dynamic Memory Allocations. #

`std::variant` and `std::optional` don't require any extra memory allocations but this is because they know which type (or types) will be stored in the object. `std::any` does not know which types might be stored and that's why it might use some heap memory.

Will it always happen, or sometimes? What are the rules? Will it happen even for a simple type like `int` ?

## Standard Ways #

Let's see what the standard says 23.8.3 [any.class]:

From The Standard:

> Implementations should avoid the use of dynamically allocated memory for a small contained value.

Example: where the object constructed is holding only an `int`. Such small-object optimization shall only be applied to types `T` for which

To sum up, implementations are encouraged to use SBO (**Small Buffer Optimisation**). But that also comes at some cost: it will make the type larger to fit the buffer.

Let's check the size of `std::any`:

Here are the results from the three compilers:

| Compiler | sizeof(any) |
| --- | --- |
| GCC 8.1 (Coliru) | 16 |
| Clang 7.0.0 (Wandbox) | 32 |
| MSVC 2017 15.7.0 32-bit | 40 |
| MSVC 2017 15.7.0 64-bit | 64 |

In general, as you see, `std::any` is not a "simple" type and it brings a lot of overhead with it. It's usually not small - due to SBO - it takes 16 or 32 bytes (GCC, Clang, or even 64 bytes in MSVC).

See below:

```
void* operator new(std::size_t count) {
    std::cout << " allocating: " << count << " bytes" << std::endl;
    return malloc(count);
}

void operator delete(void* ptr) noexcept {
    std::puts("global op delete called");
    std::free(ptr);
}

template <int Num>
class Container {
    int _array[Num];
};

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

```
        std::cout << sizeof(Container<2>) << '\n';
        std::any t(Container<2>{});
    }
    {
        std::cout << sizeof(Container<3>) << '\n';
        std::any t(Container<3>{});
    }
}
```

Moving on to the next lesson we will discuss some more practical examples.