Some lesser-known powers of std::optional

devblogs.microsoft.com/oldnewthing/20211004-00

October 4, 2021

Raymond Chen

C++17 introduced `std::optional<T>` which lets you augment the values of a type `T` with a bonus value known as `std::nullopt` which semantically represents the absence of a value. A `std::optional` which holds the value `std::nullopt` is known as *empty*.

The basic operations on `std::optional` are

- Checking if it holds a value (`has_value()`)
- Retrieving the value (`value()`)
- Assigning a value (`=`)
- Clearing the value and returning to the empty state (`reset()`)

There are other lesser-known powers of the `std::optional` .

Contextual conversion

If used in places where the language expects a Boolean (as the controlling expression for `if` , `while` , `for` , `?:` , or on either side of a `||` or `&&`), a `std::optional` is truthy if is has a value and falsy if it is empty.

```
if (opt)
```

is the same as

```
if (opt.has_value())
```

Note that this does not test whether the wrapped value is falsy.

```
std::optional<bool> opt1 = false;
if (opt1) {
    // this executes because the variable
    // is non-empty (even though it is false)
}

std::optional<void*> opt2 = nullptr;
if (opt2) {
    // this executes because the variable
    // is non-empty (even though it is nullptr)
}
```

My opinion: If `T` is itself contextually convertible to `bool`, write out `opt.has_value()` explicitly to avoid confusion.

Equality comparison against a value

An empty `std::optional<T>` compares unequal to any `T`.

```
std::optional<int> opt;
if (opt == 0) {
    // does not execute because the variable is empty
    // and is not equal to any integer.
}
```

My opinion: Use this instead of the more verbose `if (opt.has_value() && opt.value() == 0)`.

Ordering comparison against a value

An empty `std::optional` compares less than any non-empty `std::optional`, and also less than any value.

```
std::optional<int> opt;
if (opt > 0) {
    // does not execute because "empty" is
    // less than all values
}
```

My opinion: Avoid except when sorting, because this behavior differs from `NaN` (another popular “There’s nothing useful here” value) in that the corresponding opposite-sense test *does* execute.

```
if (opt <= 0) {
    // executes because "empty" is less than all values
}
```

Instead, write it out as

```
if (opt.has_value() && *opt > 0)
// or
if (opt.has_value() && *opt < 0)
```

Note that `opt.value()` and `*opt` both return the wrapped value but have different failure modes. The explicit `opt.value()` call will throw a `std::bad_optional_access` exception if the object is empty, whereas the `*opt` bypasses the verification and you get undefined behavior if the object turns out to be empty after all. In the above case, you can write the code equivalent as

```
if (opt.has_value() && opt.value() > 0)
// or
if (opt.has_value() &&amp; opt.value() < 0)
```

because the compiler can optimize out the redundant emptiness test.

Raymond Chen

Follow