# Ambiguity and Complex Types

What if we want to create objects of complex types? What if there's ambiguity when you initialize std::variant? Read below to find out the answers.

WE'LL COVER THE FOLLOWING



## Let's Discuss Ambiguity

What if you have an initialization like:

```
std::variant<int, float> intFloat { 10.5 }; // conversion from double?
```

The value **10.5** could be converted to `int` or `float`, and the compiler doesn't know which conversion should be applied. It might report a few pages of compiler errors, however.

But you can easily handle such errors by specifying which type you'd like to create:

```
std::variant<int, float> intFloat { std::in_place_index<0>, 10.5f };
// or
std::variant<int, float> intFloat { std::in_place_type<int>, 10.5f };
```

## Let's Discuss Complex Types

Similarly to `std::optional`, if you want to efficiently create objects that require several constructor arguments - then use `std::in_place_index` or `std::in_place_type`:

For example:

For example:

```
std::variant<std::vector<int>, std::string> vecStr
{
  std::in_place_index<0>, { 0, 1, 2, 3 } // initializer list passed into vector
};
```

## Unwanted Type Conversions And Narrowing #

`std::variant` in the first implementations used regular C++ rules for converting constructors and assignment operator. In a case when a conversion was required, the compiler preferred narrowing conversions which might not be what you expected.

For example:

```
std::variant<std::string, int, bool> vStrIntBool = "Hello World";
```

The above line created a variant with `bool` as the active type, not `std::string`.

The string literal `"Hello World"` is not the exact type that appears in `vStrIntBool`, so the conversion has to happen. The compiler sees two possible conversions: one from `const char*` into `bool` and then from `const char*` into `std::string`. Since `bool` is the built-in type, the compiler will select it.

There's another case with narrowing conversions:

```
variant<float, long, double> v = 0;
```

Before the fix, this line won't compile (we have several narrowing conversions possible), but after the improvement, it will hold `long`.

The implementation was improved through the fix from P0608: A sane variant converting constructor and is ready since GCC 10.0.

Below you can see a table that shows what type will be selected for a given expression, before and after the fix (P0608):

| Expression | Before Fix | After Fix |
|---|---|---|

| | | |
|---|---|---|
| 1. `variant<bool, string> v = "Hello"` | `bool` | `string` |
| 2. `variant<float, optional<double>> x = 10.05` | `float` | `optional` |
| 3. `variant<float, char> v = 0` | ill-formed | ill-formed |
| 4. `variant<float, long> v = 0` | ill-formed | selects `long` |

*Notes:*

1. The narrowing `bool` conversion is not taken into account now, and `string` is selected
2. Prefers non-narrowing conversion into `std::optional`
3. Both narrowing conversions required so the whole expression won't compile
4. before the fix the two conversions were possible after the fix `float` is not considered

> Try to match the exact type that is available in a given `std::variant` to limit the case of unexpected conversions.

---

The next lesson further elaborates on changing the current value of a variant. Read on to find out more.