# Providing Structured Binding Interface for Custom Class

To work with custom classes, we need three things:
1. std::tuple_size
2. get<N>
3. std::tuple_element

You can provide Structured Binding support for a custom class.

To do that you have to define `get<N>`, `std::tuple_size` and `std::tuple_element` specialisations for your type.

For example, if you have a class with three members, but you'd like to expose only its public interface:

```
class UserEntry {
public:
    void Load() { }

    std::string GetName() const { return name; }
    unsigned GetAge() const { return age; }
private:
    std::string name;
    unsigned age { 0 };
    size_t cacheEntry { 0 }; // not exposed
};
```

The interface for Structured Bindings:

```
// with if constexpr:
template <size_t I> auto get(const UserEntry& u) {
  if constexpr (I == 0) return u.GetName();
  else if constexpr (I == 1) return u.GetAge();
}
namespace std {
  template <> struct tuple_size<UserEntry> : std::integral_constant<size_t, 2> { };

  template <> struct tuple_element<0,UserEntry> { using type = std::string; };
  template <> struct tuple_element<1,UserEntry> { using type = unsigned; };
}
```

`tuple_size` specifies how many fields are available, `tuple_element` defines the type for a specific element and `get<N>` returns the values.

Alternatively, you can also use explicit `get<>` specializations rather than `if constexpr`:

```
template<> string get<0>(const UserEntry &u) { return u.GetName(); }

template<> unsigned get<1>(const UserEntry &u) { return u.GetAge(); }
```

For a lot of types, writing two (or several) functions might be more straightforward than using `if constexpr`.

Now you can use `UserEntry` in a structured binding, for example:

```
UserEntry u;
u.Load();
auto [name, age] = u; // read access
std:: cout << name << ", " << age << '\n';
```

Here is the complete code for you to execute (with the above code highlighted):

```
#include <iostream>
#include <tuple>

class UserEntry {
public:
    UserEntry() { }

    void Load() {
        // simulate loading from db...
        name="John";
        age=45;
        cacheEntry = 10;
    }

    std::string GetName() const { return name; }
    unsigned GetAge() const { return age; }

private:
    std::string name;
    unsigned age { 0 };
    size_t cacheEntry { 0 };
};
```

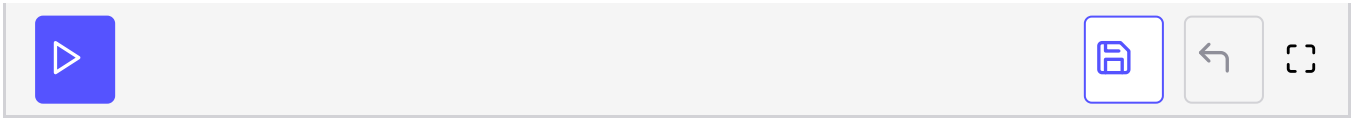
```
// read access:
template <size_t I> auto get(const UserEntry& u) {
    if constexpr (I == 0) return u.GetName();

    else if constexpr (I == 1) return u.GetAge();
}

namespace std {
    template <> struct tuple_size<UserEntry> : std::integral_constant<size_t, 2> { };

    template <> struct tuple_element<0,UserEntry> { using type = std::string; };
    template <> struct tuple_element<1,UserEntry> { using type = unsigned; };
}

int main () {
    UserEntry u;
    u.Load();
    auto [name, age] = u;
    std:: cout << name << ", " << age << '\n';
}
```



This example only allows read access of the class. If you want write access, then the class should also provide accessors that return references to members. Later you have to implement get with references support.

As you've seen `if constexpr` was used to implement `get<N>` functions, read more in the if constexpr chapter.

> *Extra Info:* The change was proposed in: P0217(wording), P0144(reasoning and examples), P0615(renaming "decomposition declaration" with "structured binding declaration").

---

We've covered the topic of structured bindings. Next up, we have the alterations made to `if` and `switch` statements.