C Interfaces For C++ Projects

Nick Deguillaume: nick@riskpath.co.uk

https://www.riskpath.co.uk/presentations/windowsFileSystem.pdf

10th April 2025

ACCU - Bristol

Motivation

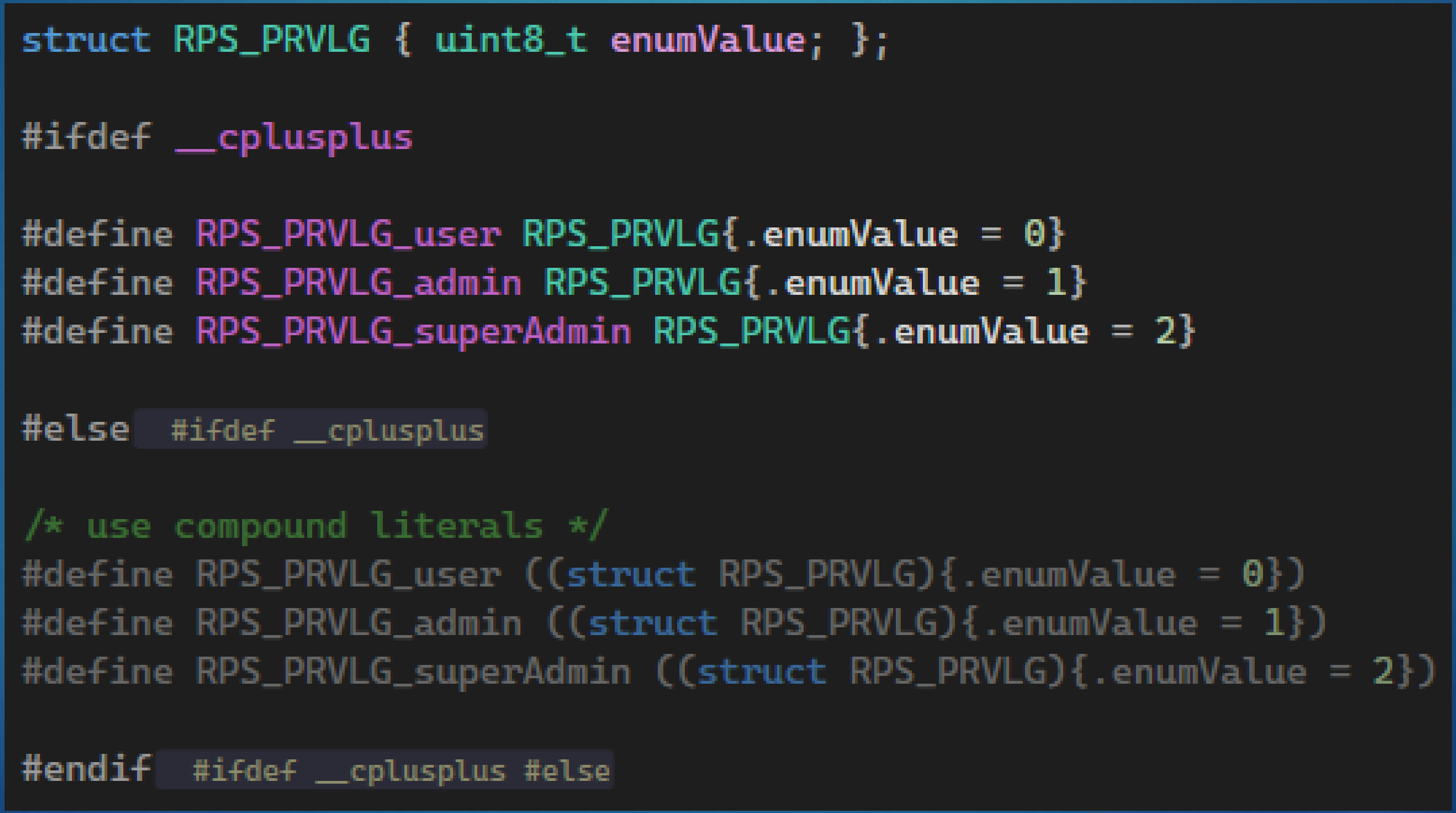
- C is the “Lingua Franca”
- If done right, gives a clean interface and good separation of concerns.

Problem

- C is not very typesafe without considerable effort.

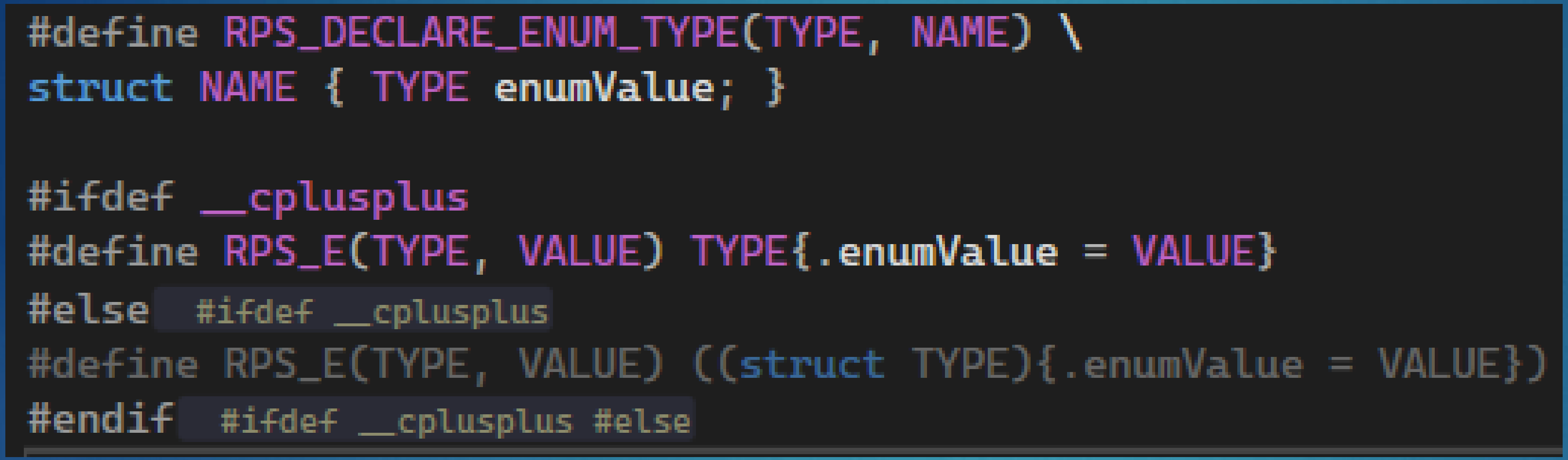
Pseudo Enumerations

- **Problem:** C enums are not typesafe and C does not provide C++'s enum class.
- **Solution:** Use structs and macros to define pseudo enumerations.

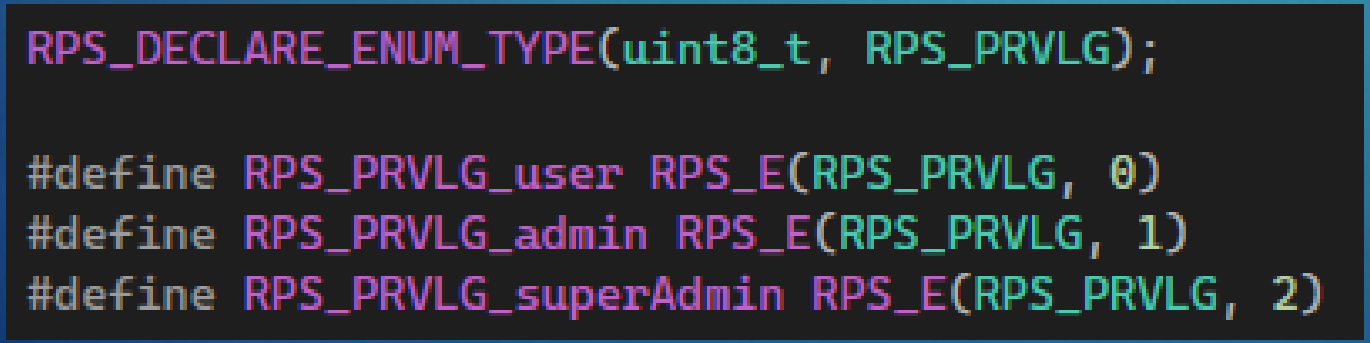


```
struct RPS_PRVLG { uint8_t enumValue; };

#ifdef __cplusplus

#define RPS_PRVLG_user RPS_PRVLG{.enumValue = 0}
#define RPS_PRVLG_admin RPS_PRVLG{.enumValue = 1}
#define RPS_PRVLG_superAdmin RPS_PRVLG{.enumValue = 2}

#else  #ifdef __cplusplus

/* use compound literals */
#define RPS_PRVLG_user ((struct RPS_PRVLG){.enumValue = 0})
#define RPS_PRVLG_admin ((struct RPS_PRVLG){.enumValue = 1})
#define RPS_PRVLG_superAdmin ((struct RPS_PRVLG){.enumValue = 2})

#endif  #ifdef __cplusplus #else
```

Pseudo Enumerations - Compact

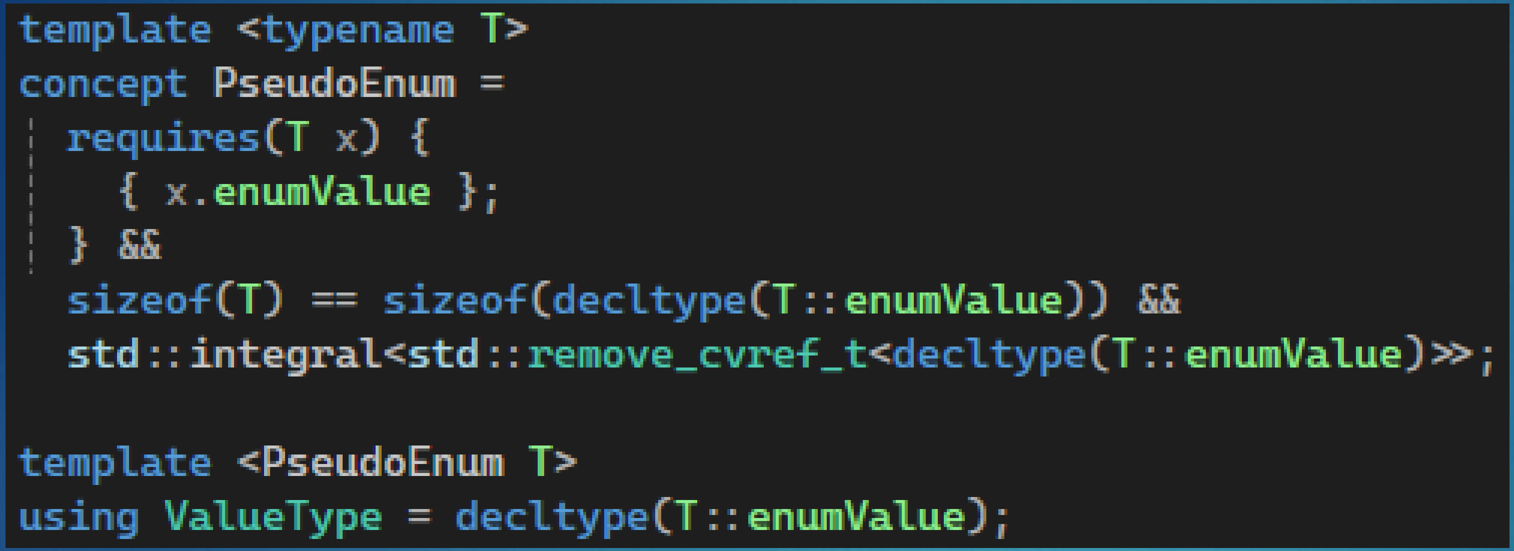


```
#define RPS_DECLARE_ENUM_TYPE(TYPE, NAME) \
struct NAME { TYPE enumValue; }

#ifdef __cplusplus
#define RPS_E(TYPE, VALUE) TYPE{.enumValue = VALUE}
#else  #ifdef __cplusplus
#define RPS_E(TYPE, VALUE) ((struct TYPE){.enumValue = VALUE})
#endif  #ifdef __cplusplus #else
```

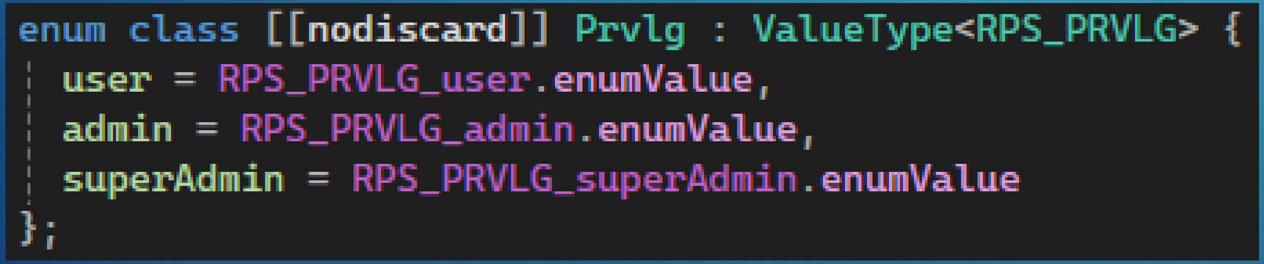


```
RPS_DECLARE_ENUM_TYPE(uint8_t, RPS_PRVLG);

#define RPS_PRVLG_user RPS_E(RPS_PRVLG, 0)
#define RPS_PRVLG_admin RPS_E(RPS_PRVLG, 1)
#define RPS_PRVLG_superAdmin RPS_E(RPS_PRVLG, 2)
```

Pseudo Enumerations - Concept



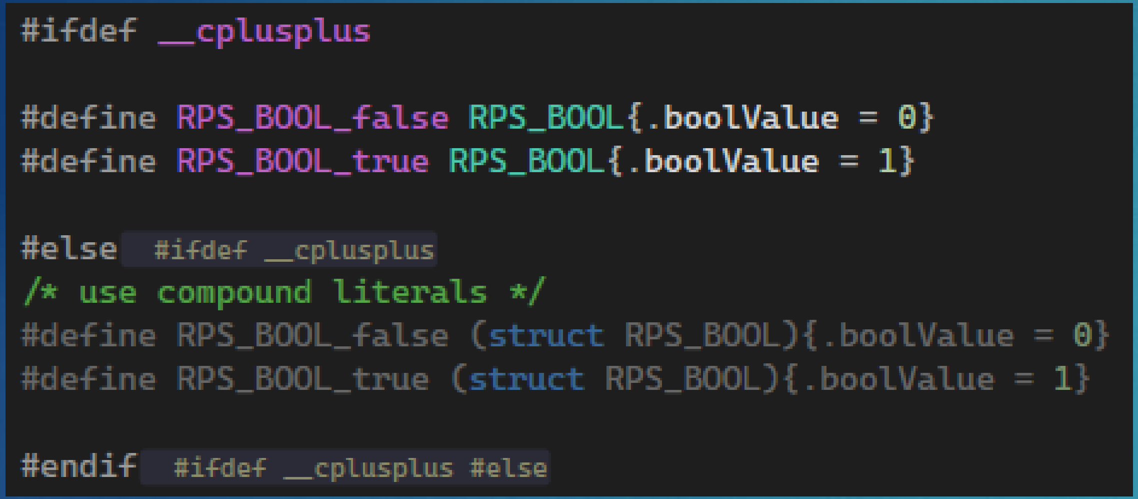
```cpp
template <typename T>
concept PseudoEnum =
  requires(T x) {
    { x.enumValue };
  } &&
  sizeof(T) == sizeof(decltype(T::enumValue)) &&
  std::integral<std::remove_cvref_t<decltype(T::enumValue)>>;

template <PseudoEnum T>
using ValueType = decltype(T::enumValue);
```



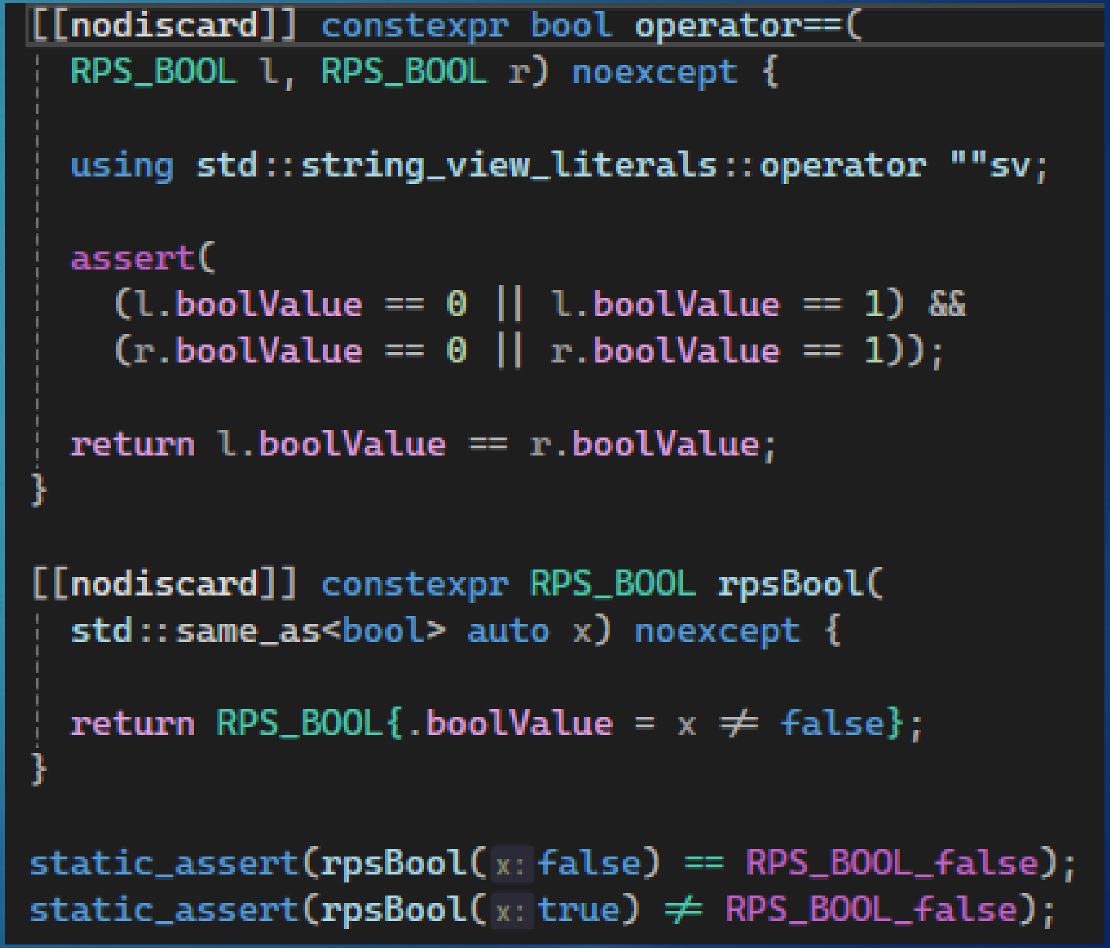
```cpp
enum class [[nodiscard]] Prvlg : ValueType<RPS_PRVLG> {
  user = RPS_PRVLG_user.enumValue,
  admin = RPS_PRVLG_admin.enumValue,
  superAdmin = RPS_PRVLG_superAdmin.enumValue
};
```

Bool

- **Problem:** C's _Bool and C++'s bool do not always play together nicely.
- **Solution:** Similar to Enum



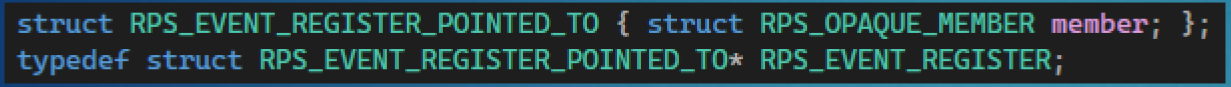
```
#ifdef __cplusplus

#define RPS_BOOL_false RPS_BOOL{.boolValue = 0}
#define RPS_BOOL_true RPS_BOOL{.boolValue = 1}

#else  #ifdef __cplusplus
/* use compound literals */
#define RPS_BOOL_false (struct RPS_BOOL){.boolValue = 0}
#define RPS_BOOL_true (struct RPS_BOOL){.boolValue = 1}

#endif  #ifdef __cplusplus #else
```



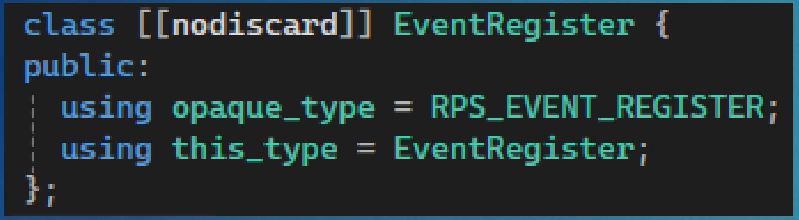
```
[[nodiscard]] constexpr bool operator==(
  RPS_BOOL l, RPS_BOOL r) noexcept {

  using std::string_view_literals::operator ""sv;

  assert(
    (l.boolValue == 0 || l.boolValue == 1) &&
    (r.boolValue == 0 || r.boolValue == 1));

  return l.boolValue == r.boolValue;
}

[[nodiscard]] constexpr RPS_BOOL rpsBool(
  std::same_as<bool> auto x) noexcept {

  return RPS_BOOL{.boolValue = x ≠ false};
}

static_assert(rpsBool(x: false) == RPS_BOOL_false);
static_assert(rpsBool(x: true) ≠ RPS_BOOL_false);
```

C++ Classes

- **Problem:** C++ classes cannot be used in C.
- **Solution:** Use opaque types.

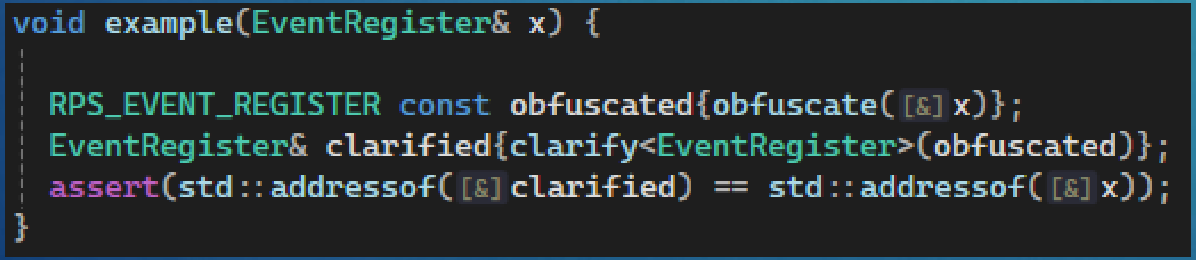


```
struct RPS_EVENT_REGISTER_POINTED_TO { struct RPS_OPAQUE_MEMBER member; };
typedef struct RPS_EVENT_REGISTER_POINTED_TO* RPS_EVENT_REGISTER;
```



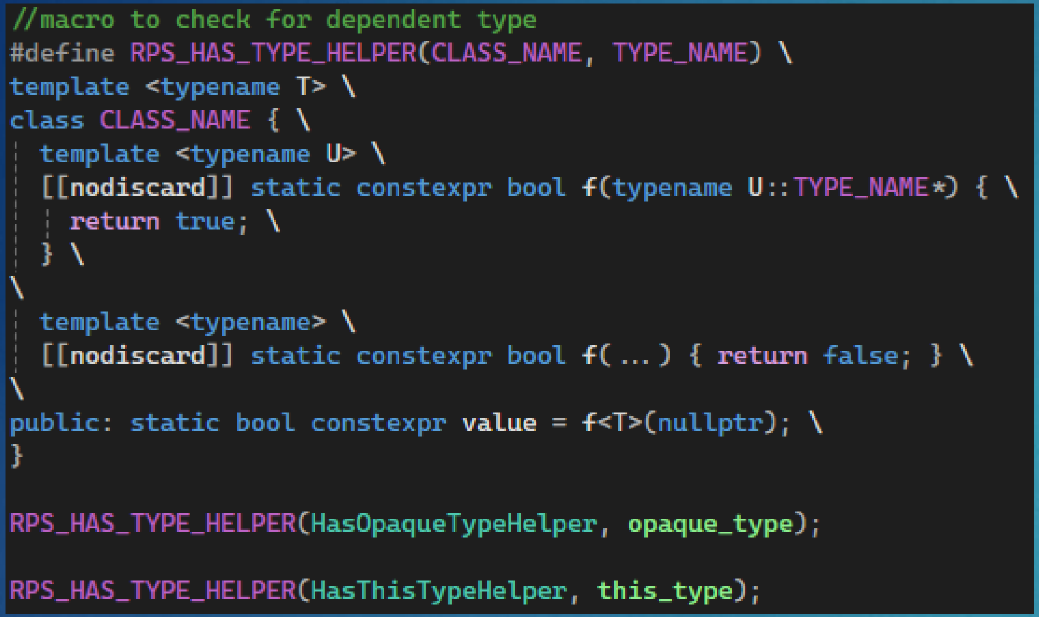
```
class [[nodiscard]] EventRegister {
public:
    using opaque_type = RPS_EVENT_REGISTER;
    using this_type = EventRegister;
};
```

We want to be abe to convert between the C++ type: **EventRegister** and the C type: **RPS_EVENT_REGISTER** in a type safe way using the functions **obfuscate** and **clarify**.

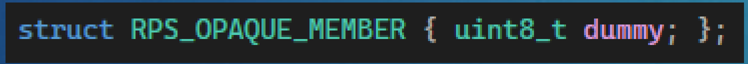


```
void example(EventRegister& x) {

    RPS_EVENT_REGISTER const obfuscated{obfuscate([&] x)};
    EventRegister& clarified{clarify<EventRegister>(obfuscated)};
    assert(std::addressof([&] clarified) == std::addressof([&] x));
}
```

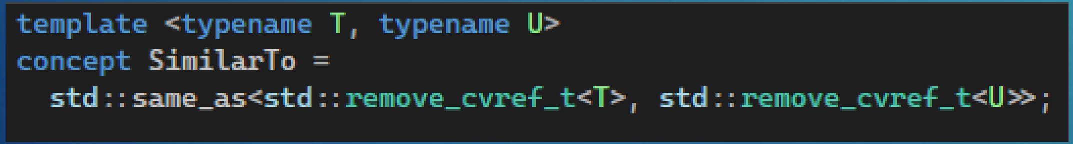
C++ Classes: Obfuscate/Clarify Implementation



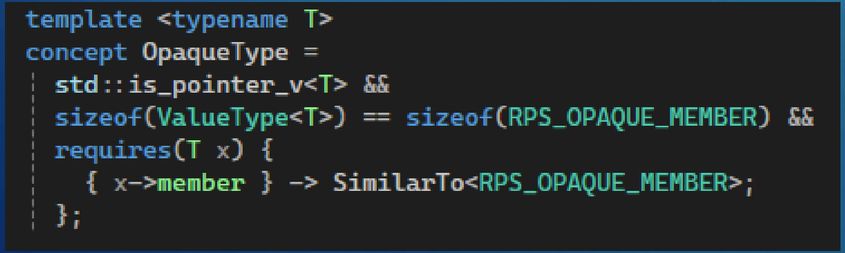
```cpp
//macro to check for dependent type
#define RPS_HAS_TYPE_HELPER(CLASS_NAME, TYPE_NAME) \
template <typename T> \
class CLASS_NAME { \
  template <typename U> \
  [[nodiscard]] static constexpr bool f(typename U::TYPE_NAME*) { \
    return true; \
  } \
\
  template <typename> \
  [[nodiscard]] static constexpr bool f( ... ) { return false; } \
\
public: static bool constexpr value = f<T>(nullptr); \
}

RPS_HAS_TYPE_HELPER(HasOpaqueTypeHelper, opaque_type);

RPS_HAS_TYPE_HELPER(HasThisTypeHelper, this_type);
```



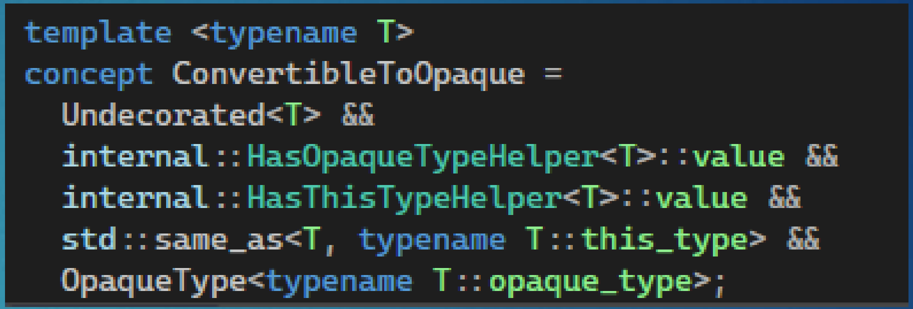
```cpp
struct RPS_OPAQUE_MEMBER { uint8_t dummy; };
```



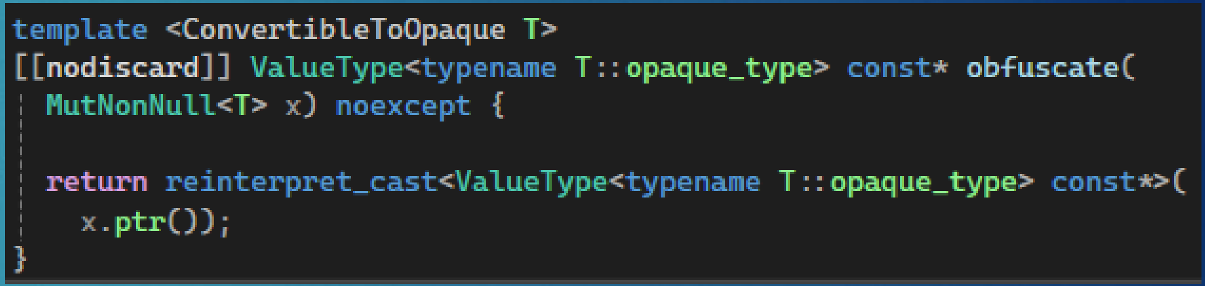
```cpp
template <typename T, typename U>
concept SimilarTo =
  std::same_as<std::remove_cvref_t<T>, std::remove_cvref_t<U>>;
```



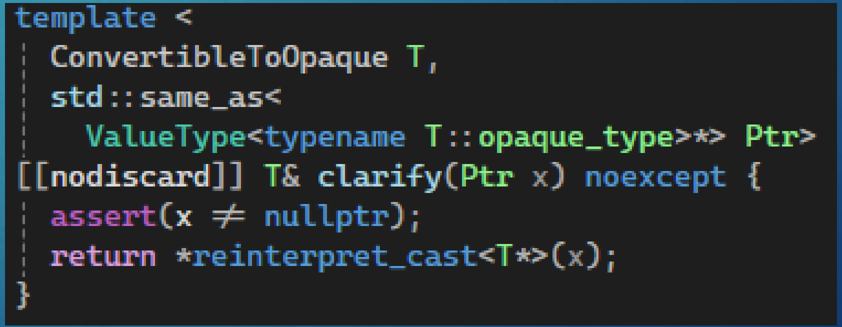
```cpp
template <typename T>
concept OpaqueType =
  std::is_pointer_v<T> &&
  sizeof(ValueType<T>) == sizeof(RPS_OPAQUE_MEMBER) &&
  requires(T x) {
    { x->member } -> SimilarTo<RPS_OPAQUE_MEMBER>;
  };
```



```cpp
template <typename T>
concept ConvertibleToOpaque =
  Undecorated<T> &&
  internal::HasOpaqueTypeHelper<T>::value &&
  internal::HasThisTypeHelper<T>::value &&
  std::same_as<T, typename T::this_type> &&
  OpaqueType<typename T::opaque_type>;
```



```cpp
template <ConvertibleToOpaque T>
[[nodiscard]] ValueType<typename T::opaque_type> const* obfuscate(
  MutNonNull<T> x) noexcept {

  return reinterpret_cast<ValueType<typename T::opaque_type> const*>(
    x.ptr());
}
```



```cpp
template <
  ConvertibleToOpaque T,
  std::same_as<
    ValueType<typename T::opaque_type>*> Ptr>
[[nodiscard]] T& clarify(Ptr x) noexcept {
  assert(x ≠ nullptr);
  return *reinterpret_cast<T*>(x);
}
```