SuperCharge Your IPC Programs With C++20 and CCI Pattern

ARIAN AJDARI

Rules

- **IPC** stands for Intra-Process Communication
- **Programs** stand for software running in a constrained environment
- **CCI** stands for:
  - **C**ontract
  - **C**oncept
  - **I**mplementation

Problem Definition: Channel Extraction

1. Images are used throughout different applications.
2. We want to extract color channels that images consist of.
3. We want to provide a service with communication endpoint.

Implementation: Standard IP

```c
sd_bus_error error = SD_BUS_ERROR_NULL;
sd_bus_message *m = NULL;
sd_bus *bus = NULL;
const int32_t result;
int r;
```

Implementation: Standard IP

```
const char* file = "/home/sample";
const char* save_path = "/home/modified";
const uint8_t channel = 1;
```

Implementation: Standard IP

```
r = sd_bus_call_method(
 bus,
 "com.cppcon2024.Channel",
 "/com/cppcon2024/Channel",
 "com.cppcon2024.Channel",
 "ExtractChannel",
 &error, &m, "ooy",
 file, save_path, channel);
```

Solution

- Create a CONTRACT which specifies the characteristics of a data-type.
- Create a CONCEPT which enforces the CONTRACT
- Create a IMPLEMENTATION constrained by CONCEPT.

SD_BUS Contract

```
typedef std::unique_ptr<
  sd_bus,
  decltype(&sd_bus_unref)> SD_BUS_INNER_TYPE;

struct SD_BUS_CONTRACT {
template<typename T> static constexpr bool IsUniquePtr =
  std::is_same<T, SD_BUS_INNER_TYPE>::value;
};
```

SD_BUS Concept

```
template <typename T>
concept SD_BUS_CONCEPT =
    requires { requires SD_BUS_CONTRACT::IsUniquePtr<T>; };
```

SD_BUS Implementation

```
template<SD_BUS_CONCEPT T = SD_BUS_INNER_TYPE>
[[nodiscard]] T MAKE_SD_BUS()
{
    return T(nullptr, &sd_bus_unref);
}
```

SD_BUS Implementation

```
template<SD_BUS_CONCEPT T = SD_BUS_INNER_TYPE>
struct SD_BUS
{
    T bus = MAKE_SD_BUS();
    decltype(bus.get()) ptr = bus.get();
    decltype(bus.get())* addressof_ptr = &ptr;

    inline auto operator()()
    {
        return addressof_ptr;
    }

    inline auto operator decltype(auto)() const
    {
        return ptr;
    }
};
```

SD_BUS Implementation

```
template<typename T> concept SD_BUS_TYPE_CONCEPT =
    std::is_same<T, SD_BUS<SD_BUS_INNER_TYPE>>::value;
```

SD_BUS_MESSAGE Contract

```
typedef std::unique_ptr<
  sd_bus_message,
  decltype(&sd_bus_message_unref)> SD_BUS_MESSAGE_TYPE;

struct SD_BUS_MESSAGE_CONTRACT {
  template<typename T> static constexpr bool IsUniquePtr =
    std::is_same<T, SD_BUS_MESSAGE_TYPE>::value; };
```

SD_BUS_MESSAGE Concept

```
template <typename T>
concept SD_BUS_MESSAGE_CONCEPT =
  requires { requires SD_BUS_MESSAGE_CONTRACT::IsUniquePtr<T>; };
```

SD_BUS_CONCEPT Implementation

```
template<SD_BUS_MESSAGE_CONCEPT T = SD_BUS_MESSAGE_TYPE>
[[nodiscard]] T MAKE_SD_BUS_MESSAGE()
{
    return T(nullptr, &sd_bus_message_unref);
}
```

SD_BUS_MESSAGE Implementation

```
template<SD_BUS_MESSAGE_CONCEPT T = SD_BUS_MESSAGE_TYPE>
struct SD_BUS_MESSAGE {
  T message = MAKE_SD_BUS_MESSAGE();
  decltype(message.get()) ptr = message.get();
  decltype(message.get())* addressof_ptr = &ptr;

  inline auto operator()() const { return addressof_ptr; }
  inline operator decltype(auto)() const { return ptr; } };
```

SD_BUS_MESSAGE Implementation

```
template<typename T> concept SD_BUS_MESSAGE_CONCEPT =
  std::is_same<T, SD_BUS_MESSAGE<SD_BUS_MESSAGE_TYPE>>::value;
```

SD_BUS_ERROR_TYPE Contract

```
struct SD_BUS_ERROR_TYPE_CONTRACT {

  template<typename T, typename = int>
  static constexpr bool HasFunctionOperatorOverload = false;

  template<typename T>
  static constexpr bool HasFunctionOperatorOverload<
    T, decltype(&T::operator const char*, 0)> = true;

  template<typename T>
  static constexpr bool IsTriviallyConstructible =
    std::is_constructible<T>::value;
};
```

SD_BUS_ERROR_TYPE Concept

```
template <typename T> concept SD_BUS_ERROR_TYPE_CONCEPT =
  requires { requires SD_BUS_ERROR_TYPE_CONTRACT::HasFunctionOperatorOverload<T>;} &&
  requires { requires SD_BUS_ERROR_TYPE_CONTRACT::IsTriviallyConstructible<T>;   };
```

SD_BUS_ERROR_TYPE Implementation

```
class SD_BUS_ERROR_TYPE
{
public:
  operator const char*() const { return nullptr; }

private:
  friend class SD_BUS_ERROR_TYPE_CONTRACT;
};
```

SD_BUS_ERROR Contract

```cpp
class SD_BUS_ERROR_STRUCT_CONTRACT {
public:
  template <typename T, typename = int>
  static constexpr bool HasFieldName = false;

  template <typename T>
  static constexpr bool HasFieldName<T, decltype((void) T::name, 0)> = true;

  template<typename T>
  static constexpr bool IsTypeNameStr = std::is_same<decltype(T::name), std::string>::value;


  template <typename T, typename = int>
  static constexpr bool HasFieldDescription = false;

  template <typename T>
  static constexpr bool HasFieldDescription<T, decltype((void) T::description, 0)> = true;

  template<typename T> static constexpr bool IsTypeDescriptionStr =
    std::is_same<decltype(T::description), std::string>::value;
};
```

SD_BUS_ERROR Concept

```
template <typename T> concept SD_BUS_ERROR_STRUCT_CONCEPT =
  requires { requires SD_BUS_ERROR_STRUCT_CONTRACT::HasFieldName<T>         ;} &&
  requires { requires SD_BUS_ERROR_STRUCT_CONTRACT::IsTypeNameStr<T>        ;} &&
  requires { requires SD_BUS_ERROR_STRUCT_CONTRACT::HasFieldDescription<T> ;} &&
  requires { requires SD_BUS_ERROR_STRUCT_CONTRACT::IsTypeDescriptionStr<T>;};
```

SD_BUS_ERROR_TYPE Implementation

```
template <SD_BUS_ERROR_TYPE_CONCEPT T>
[[nodiscard("Return value must be placed inside variable")]]
constexpr auto make_sdbus_error_structure()
{
  return (sd_bus_error){.name = T{}, .message = T{}, ._need_free = int{}};
}
```

SD_BUS_ERROR_TYPE Implementation

```
template<SD_BUS_ERROR_TYPE_CONCEPT T = SD_BUS_ERROR_TYPE> struct SD_BUS_ERROR_STRUCT_INNER {

  SD_BUS_ERROR_STRUCT_INNER() : name(""), description("") { }

  SD_BUS_ERROR_STRUCT_INNER(
    const std::string& name,
    const std::string& description) :
      name(std::move(name)), description(std::move(description)) { }

  virtual ~SD_BUS_ERROR_STRUCT_INNER() { sd_bus_error_free(&sd_bus_error); }

  std::string name;
  std::string description;
  sd_bus_error sd_bus_error = make_sdbus_error_structure<T>();
};
```

SD_BUS_ERROR Implementation

```cpp
template<SD_BUS_ERROR_STRUCT_CONCEPT T> struct SD_BUS_ERROR_STRUCT {
  SD_BUS_ERROR_STRUCT() : t(T()) { }

  SD_BUS_ERROR_STRUCT(
    const std::string& name,
    const std::string& description) : t(T(name, description)) { }

  [[nodiscard("Overloaded operator -> returns inner instance of sd_bus_error.")]]
  auto operator ->() { return &t.sd_bus_error; }

  auto operator()() { return &t.sd_bus_error; }

   T t;
};
```

SD_BUS_ERROR Implementation

```
typedef SD_BUS_ERROR_STRUCT<SD_BUS_ERROR_STRUCT_INNER<>> SD_BUS_ERROR;

template<typename T>
concept SD_BUS_ERROR_CONCEPT = std::is_same<T, SD_BUS_ERROR>::value;
```

SERVICE_NAME Contract

```
constexpr int SERVICE_NAME_MAX_LENGTH = 30;
constexpr const char* service_name_starts_with = "com";

struct SERVICE_NAME_CONTRACT {
  template<size_t T>
  constexpr static bool IsMaxLength = T < SERVICE_NAME_MAX_LENGTH;

  template<bool T>
  constexpr static bool StartsWith = T;
};
```

SERVICE_NAME Contract

```cpp
template<size_t N>
static constexpr size_t SERVICE_NAME_IS_MAX_LENGTH(const char (&str)[N])
{
  return N;
}
```

SERVICE_NAME Contract

```
template<std::size_t N>
static constexpr auto SERVICE_NAME_STARTS_WITH(const char (&str)[N]) {
  auto subspan = std::span{str}.first(3);

  int i= 0;
  for(auto const& elem : subspan)
    if(elem != service_name_starts_with[i++]) return false;

  return true;
};
```

SERVICE_NAME Concept

```
template<size_t T, bool U>
concept SERVICE_NAME_CONCEPT =
  requires { requires SERVICE_NAME_CONTRACT::IsMaxLength<T>;} &&
  requires { requires SERVICE_NAME_CONTRACT::StartsWith<U> ;};
```

SERVICE_NAME Implementation

```cpp
template<size_t T, bool U>
requires SERVICE_NAME_CONCEPT<T, U>
[[nodiscard]] constexpr auto SERVICE_NAME_INNER(const char (&str)[T])
{
    return SERVICE_NAME_TYPE{str};
}

#define SERVICE_NAME(T) \\
  SERVICE_NAME_INNER<SERVICE_NAME_IS_MAX_LENGTH(T),SERVICE_NAME_STARTS_WITH(T)>(T)
```

SERVICE_NAME Implementation

```
template<typename T>
concept SERVICE_NAME_TYPE_CONCEPT = std::is_same<T, SERVICE_NAME_TYPE>::value;
```

DBUS_PATH Contract

```
struct DBUS_PATH
{
  template<std::size_t N>
  constexpr explicit DBUS_PATH(const char (&value)[N]) : value(value) { }

  const char* value;
  std::string signature{"o"};
};
```

DBUS_UINT8 Contract

```
struct DBUS_UINT8
{
  constexpr explicit DBUS_UINT8(uint8_t value) : value(value) { }

  uint8_t value;
  std::string signature{"y"};
};
```

DBUS_TYPES Concept

```cpp
template<typename T, typename...U>
concept AnyOf = (std::is_same<T, U>::value || ...);

template <typename...T>
concept DBusSupportedTypes = (AnyOf<T, DBUS_PATH, DBUS_UINT8> && ...);
```

Some helper definitions

```
#define RED_CHANNEL DBUS_UINT8{0}
#define GREEN_CHANNEL DBUS_UINT8{1}
#define BLUE_CHANNEL DBUS_UINT8{2}
```

SD_BUS_CALL_METHOD reimagined

```
template<typename... DBUS_SUPPORTED_TYPES>
requires DBusSupportedTypes<DBUS_SUPPORTED_TYPES...>
[[nodiscard]] constexpr inline auto SD_BUS_CALL_METHOD(
  SD_BUS_TYPE_CONCEPT auto const& bus,
  SERVICE_NAME_TYPE_CONCEPT auto const& service_name,
  OBJECT_PATH_TYPE_CONCEPT auto const& object_path,
  INTERFACE_NAME_TYPE_CONCEPT auto const& interface_name,
  METHOD_NAME_CONCEPT auto const& method_name,
  SD_BUS_ERROR_CONCEPT auto& error,
  SD_BUS_MESSAGE_CONCEPT auto& m,
  DBUS_SUPPORTED_TYPES const&...args
);
```

SD_BUS_CALL_METHOD reimagined

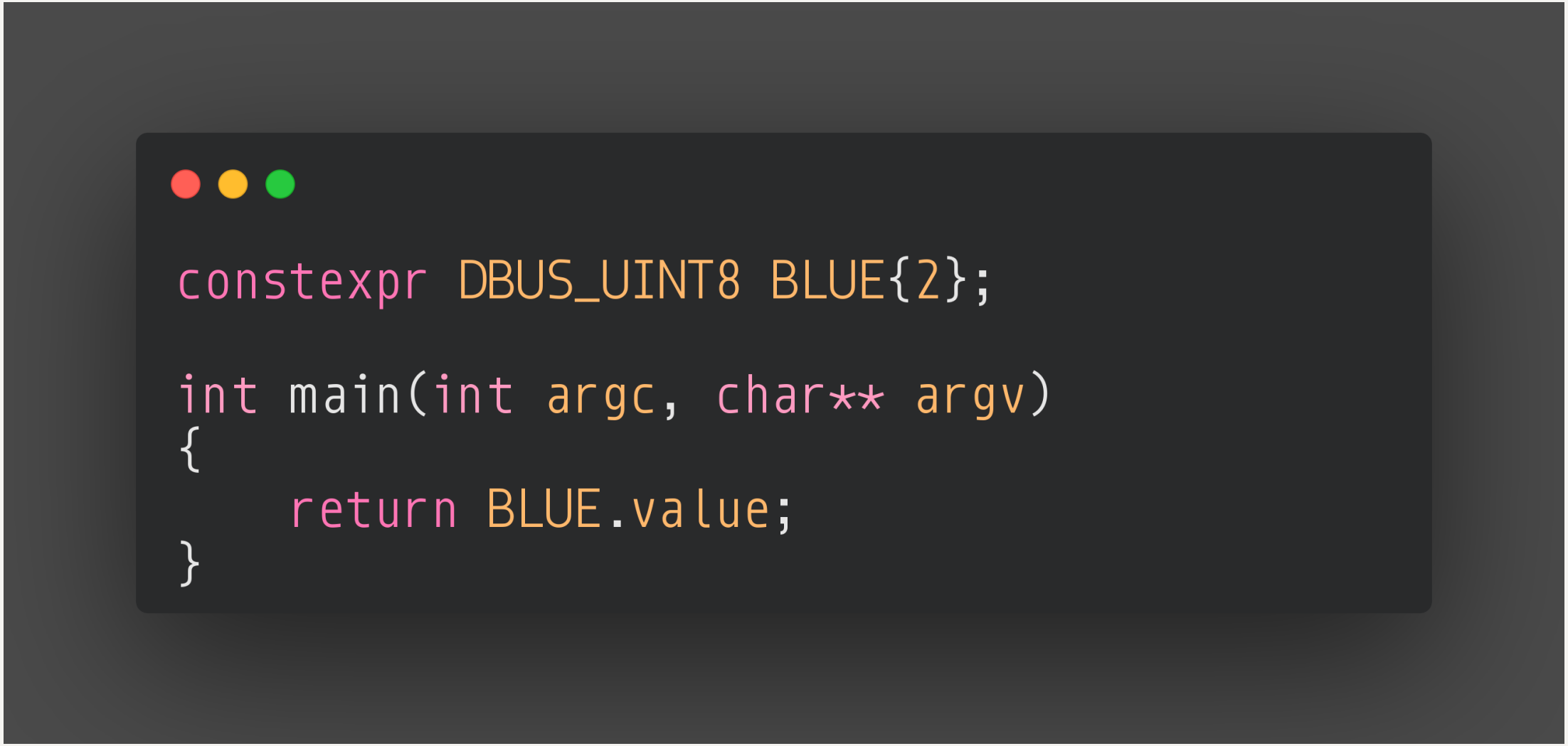
```
sd_bus_call_method(
  bus,
  service_name.str,
  object_path.str,
  interface_name.str,
  method_name.str,
  error(),
  m(),
  (args.signature + ...).c_str(),
  args.value...
);
```

Comparison

```
sd_bus_call_method(
  bus, service_name,
  object_path, interface_name,
  method_name, &error,
  &m, "ooy", file,
  save_path, channel);
```

```
sd_bus_call_method(
  bus, service_name,
  object_path, interface_name,
  method_name, error, m,
  DBUS_PATH{"/home/image/sample"},
  DBUS_PATH{"/home/image/red"},
  RED_CHANNEL);
```

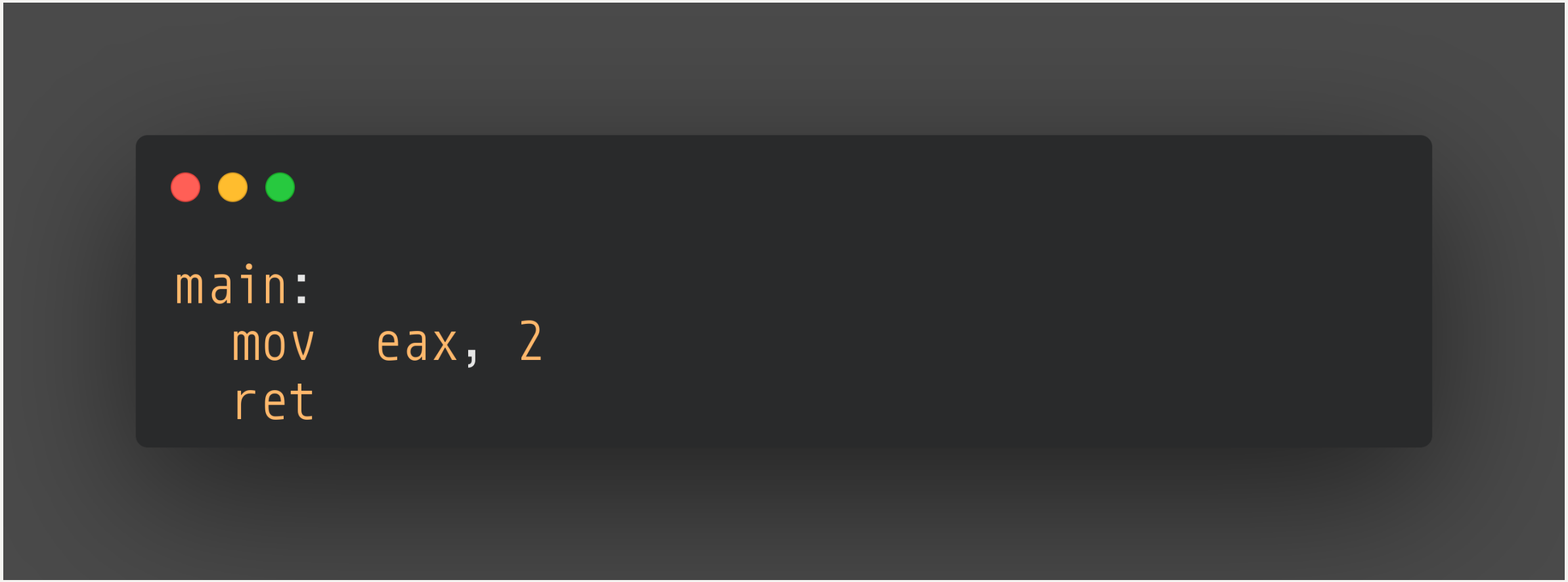
How many assembly lines?



```
constexpr DBUS_UINT8 BLUE{2};

int main(int argc, char** argv)
{
    return BLUE.value;
}
```

Assembly generated when CCI is utilized



```
main:
  mov  eax, 2
  ret
```

Designing a class from description

Design a class called PERSON with following characteristics:

1. Has name, surname, age, expenses as attributes
2. Has getters for each of attributes
3. Attribute name has minimum length of 3, maximum length of 40 and has to start with 'A'

Person: Helper Functions

```
template<bool T>
struct logical_not
{
    static constexpr bool value = !T;
};
```

Person: Helper Functions

```
class FieldChecker {
public:
    template <typename T, typename Field>
    static constexpr bool HasField(Field field) {
        return std::is_member_object_pointer<decltype(field)>::value;
    }

    template <typename T, typename... Fields>
    static constexpr bool HasFields(Fields... field) {
        return (HasField<T>(field) && ...);
    }
};
```

Person: Helper Functions

```
class FunctionChecker {
public:
    template <typename T, typename Function>
    static constexpr bool HasFunction(Function function) {
        return std::is_member_function_pointer<decltype(function)>::value;
    }

    template <typename T, typename... Functions>
    static constexpr bool HasFunctions(Functions... function) {
        return (HasFunction<T>(function) && ...);
    }
};
```

Person: Types

```
struct NAME {
  template<std::size_t N> constexpr explicit NAME(const char (&name)[N])
    : name(name) { }
  std::string name; };

struct SURNAME {
  template<std::size_t N> constexpr explicit SURNAME(const char (&surname)[N])
    : surname(surname) { }
  std::string surname; };

struct AGE {
  constexpr explicit AGE(int age) : age(age) { }
  int age; };

struct EXPENSES {
  constexpr explicit EXPENSES(double expenses) : expenses(expenses) { }
  double expenses; };
```

Person: Contract

```
template<typename T> class PersonContract
{
public:
    static constexpr bool FieldsExist =
      FieldChecker::HasFields<T>(&T::name, &T::surname, &T::age, &T::expenses);

    static constexpr bool FunctionsExist =
      FunctionChecker::HasFunctions<T>(&T::getName,&T::getSurname,&T::getAge, &T::getExpenses);

    static constexpr bool IsNotDefaultConstructible =
      logical_not<std::is_default_constructible<T>::value>::value;
};
```

Person: Concept

```
template <typename Person>
concept PersonConcept =
    PersonContract<Person>::FieldsExist &&
    PersonContract<Person>::FunctionsExist &&
    PersonContract<Person>::IsNotDefaultConstructible;
```

Person: Types

```
struct PersonTypes
{
    template<typename T, typename...U>
    static constexpr bool AnyOf = (std::is_same<T,U>::value || ...);

    template<typename...T>
    static constexpr bool SupportedTypes = (AnyOf<T, NAME, SURNAME, AGE, EXPENSES> && ...);
};
```

Person: Types

```
template<bool...U>
concept IsNameValid = (U && ...);
```

Person: Name Type

```cpp
template<std::size_t N, bool...U>
requires IsNameValid<U...>
[[nodiscard]] constexpr auto MAKE_NAME_OBJECT(const char(&str)[N]) { return NAME{str}; }

template<std::size_t N>
static constexpr bool IsMinLength = N > 3;

template<std::size_t N>
static constexpr bool IsMaxLength = N < 40;

template<std::size_t N>
static constexpr std::size_t Length(const char(&str)[N]) { return N; }

template<std::size_t N>
static constexpr bool StartsWith(const char(&str)[N]) { return str[0] == 'A'; }
```

Person: Helper Definition

```
#define MAKE_NAME(T)
            MAKE_NAME_OBJECT<Length(T)                    ,
                             IsMinLength<Length(T)>,
                             IsMaxLength<Length(T)>,
                             StartsWith(T)>(T)
```

Person: Implementation

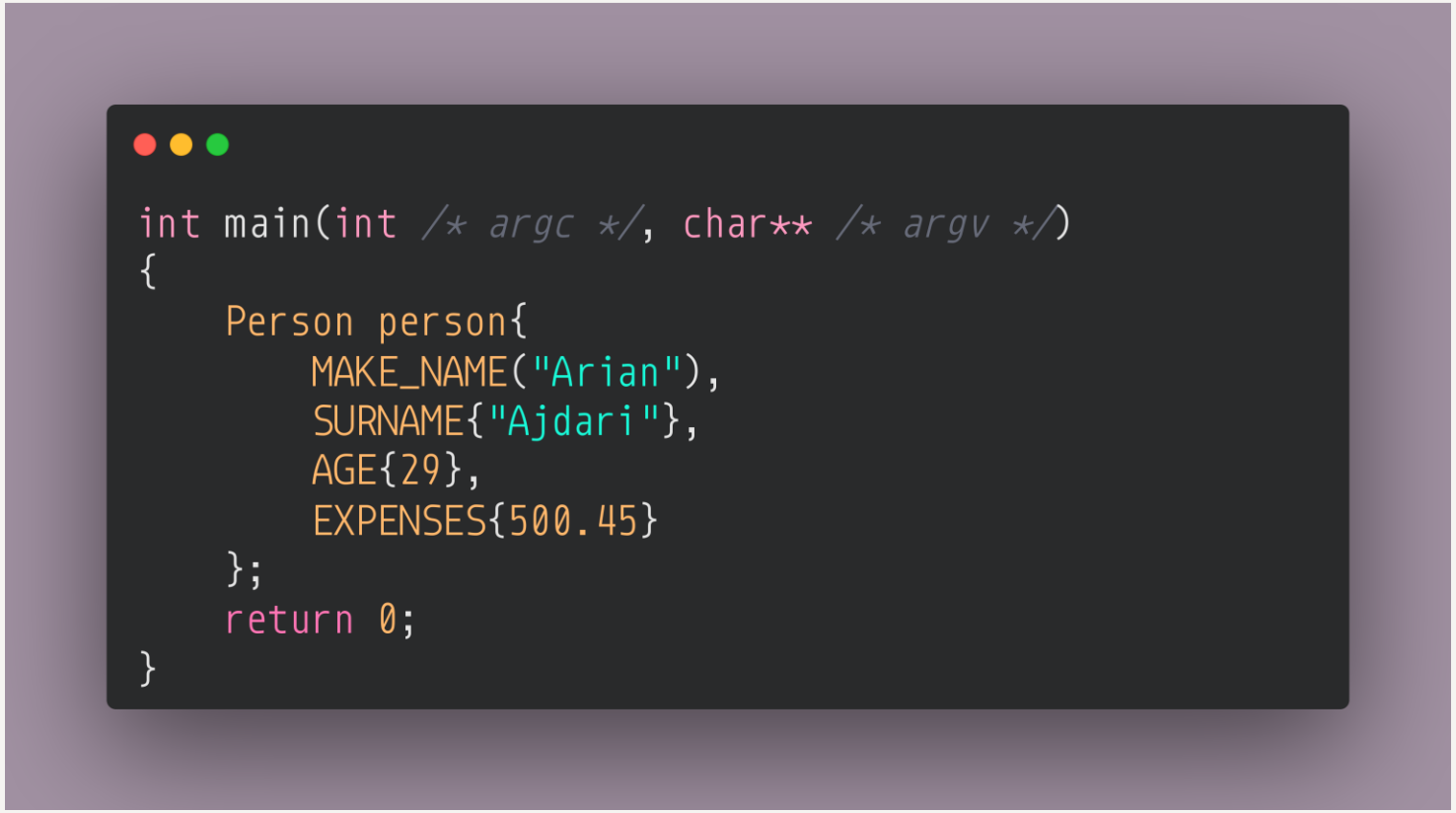
```
struct Person {
  template<typename NAME, typename SURNAME, typename AGE, typename EXPENSES>
  requires PersonTypes::SupportedTypes<NAME, SURNAME, AGE, EXPENSES>
  Person(NAME name, SURNAME surname, AGE age, EXPENSES expenses) :
      name(name), surname(surname), age(age), expenses(expenses) { }

  NAME getName() {return name; }
  SURNAME getSurname() { return surname;}
  AGE getAge() const { return age;}
  EXPENSES getExpenses() { return expenses;}

  NAME name;
  SURNAME surname;
  AGE age;
  EXPENSES expenses;

 friend class PersonContract<Person>;
};
```

Wrapping Up


int main(int /* argc */, char** /* argv */)
{
Person person{
MAKE_NAME("Arian"),
SURNAME{"Ajdari"},
AGE{29},
EXPENSES{500.45}
};
return 0;
}

Conclusion

- CCI is a powerful tool in developer's arsenal
  - It allows to shift some work from runtime-evaluation to compile-evaluation
  - It can serve to wrap low-level facilities into a higher level abstractions
- Contracts can serve to express programmatically the meaning of descriptive text of a particular software artifact, or even an UML Diagram. Therefore, it can serve as an signature of an software artifact.
- The resulting binaries are not bigger than their counterparts without CCI, because all of the evaluation takes part in compile-time.

What CCI is not:

- An universal tool that needs to be used everywhere.
- Techniques presented here can also be done with other C++ techniques.
- It falls on the responsibility of the programmer to decide where to utilize it.

I hope you enjoyed this presentation

? Questions ?