Using Modern C++ to Eliminate Virtual Functions

JONATHAN GOPEL

Cppcon
The C++ Conference
2022
September 12th-16th

When is virtual useful

- Requiring a specific interface
- Adding configurability to objects
- Holding multiple different derived types with a shared base class in a single container

Why replace virtual

- Less indirection
- Capture system properties more statically
- Greater flexibility in design
- Can improve performance
- Because we can

Things that don't count

- Recreating anything like a vtable
  - `std::vector<std::any>`
  - `std::vector<std::variant>`
- Type erasure

Binding interfaces

```cpp
struct FooInterface {
  [[nodiscard]] virtual auto func() const -> int = 0;
};

struct Foo final : public FooInterface {
  [[nodiscard]] auto func() const -> int override {
    return 42;
  }
};
```

Binding interfaces

```cpp
struct FooInterface {
  FooInterface() = default;
  FooInterface(const FooInterface&) = default;
  FooInterface(FooInterface&&) = default;
  FooInterface& operator=(const FooInterface&) = default;
  FooInterface& operator=(FooInterface&&) = default;
  virtual ~FooInterface() = default;

  [[nodiscard]] virtual auto func() const -> int = 0;
};

struct Foo final : public FooInterface {
  Foo() = default;
  Foo(const Foo&) = default;
  Foo(Foo&&) = default;
  Foo& operator=(const Foo&) = default;
  Foo& operator=(Foo&&) = default;
  virtual ~Foo() = default;

  [[nodiscard]] auto func() const -> int override {
    return 42;
  }
};
```

Binding interfaces

```cpp
template <typename T>
concept CFoo = requires(T foo) {
  { foo.func() } -> std::same_as<int>;
};

struct Foo {
  [[nodiscard]] auto func() const -> int {
    return 42;
  }
};

static_assert(CFoo<Foo>);
```

Binding interfaces

```cpp
template <typename T>
concept CFoo = requires(T foo) {
  { foo.func() } -> std::integral;
};

struct Foo {
  [[nodiscard]] auto func() const -> int {
    return 42;
  }
};

static_assert(CFoo<Foo>);
```

Binding interfaces

```cpp
// with virtual
std::unique_ptr<FooInterface> foo = std::make_unique<Foo>();
auto func(std::unique_ptr<FooInterface> foo2) {
  // Implementation here
}

// without virtual
Foo foo{};
auto func(CFoo auto& foo2) {
  // Implementation here
}
```

Owning a polymorphic type

```cpp
class Bar {
public:
  constexpr Bar(std::unique_ptr<FooInterface> input_foo)
    : foo{std::move(input_foo)} {}

private:
  std::unique_ptr<FooInterface> foo{};
};
```

Owning a polymorphic type

```
template <typename TFoo>
class Bar {
public:
  constexpr Bar(TFoo input_foo)
    : foo{input_foo} {}

private:
  TFoo foo{};
};
```

Owning a polymorphic type

```cpp
template <CFoo TFoo>
class Bar {
public:
  constexpr Bar(TFoo input_foo)
    : foo{input_foo} {}

private:
  TFoo foo{};
};
```

Owning a polymorphic type

```cpp
class Bar {
public:
  constexpr Bar(std::unique_ptr<FooInterface> input_foo)
    : foo{std::move(input_foo)} {}

private:
  std::unique_ptr<FooInterface> foo{};
};
```

Owning a polymorphic type

```cpp
class Bar {
public:
  constexpr Bar(std::unique_ptr<FooInterface> input_foo)
    : foo{std::move(input_foo)} {}

  constexpr auto set_foo(std::unique_ptr<FooInterface> input_foo) {
    foo = std::move(input_foo);
  }

private:
  std::unique_ptr<FooInterface> foo{};
};
```

Owning a polymorphic type

```
template <CFoo TFoo>
class Bar {
public:
  constexpr Bar(TFoo input_foo)
    : foo{input_foo} {}

private:
  TFoo foo{};
};
```

Owning a polymorphic type

```cpp
template <CFoo... TFoos>
class Bar {
public:
  constexpr Bar(auto input_foo)
    : foo{input_foo} {}

  constexpr auto set_foo(auto input_foo) -> void {
    foo = input_foo;
  }

private:
  std::variant<TFoos...> foo{};
};
```

Owning a polymorphic type

```cpp
template <CFoo... TFoos>
class Bar {
public:
  constexpr Bar(CFoo auto input_foo)
    : foo{input_foo} {}

  constexpr auto set_foo(CFoo auto input_foo) -> void {
    foo = input_foo;
  }

private:
  std::variant<TFoos...> foo{};
};
```

Owning a polymorphic type

```cpp
template <CFoo... TFoos>
class Bar {
public:
  constexpr Bar(auto input_foo)
    : foo{input_foo} {}

  constexpr auto set_foo(auto input_foo) -> void {
    foo = input_foo;
  }

private:
  std::variant<TFoos...> foo{};
};
```

Owning a polymorphic type

```
template<typename T, typename... Ts>
concept same_as_any = (... or std::same_as<T, Ts>);

template <CFoo... TFoos>
class Bar {
public:
  constexpr Bar(auto input_foo)
    : foo{input_foo} {}

  constexpr auto set_foo(auto input_foo) -> void {
    foo = input_foo;
  }

private:
  std::variant<TFoos...> foo{};
};
```

Owning a polymorphic type

```cpp
template<typename T, typename... Ts>
concept same_as_any = (... or std::same_as<T, Ts>);

template <CFoo... TFoos>
class Bar {
public:
  constexpr Bar(same_as_any<TFoos...> auto input_foo)
    : foo{input_foo} {}

  constexpr auto set_foo(same_as_any<TFoos...> auto input_foo) -> void
    foo = input_foo;
  }

private:
  std::variant<TFoos...> foo{};
};
```

Owning a polymorphic type

```cpp
// with virtual
Bar bar{std::make_unique<Foo>()};

// without virtual
Bar bar{Foo{}};
Bar<Foo1, Foo2> bar{Foo1{}};
```

Storing multiple types

```cpp
class Baz {
public:
  auto store(std::unique_ptr<FooInterface> value) -> void {
    data.push_back(std::move(value));
  }

private:
  std::vector<std::unique_ptr<FooInterface>> data{};
};
```

Desired properties

- List of all the types that might be stored
- Container that can hold many different types simultaneously
- Container that can hold multiple objects of a single type

Storing multiple types

```
template <typename... TFoos>
class Baz {
public:
  template <typename T>
  auto store(T value) {
    return std::get<std::vector<T>>(data).push_back(value);
  }

private:
  std::tuple<std::vector<TFoos>...> data{};
};
```

Storing multiple types

```cpp
template <CFoo... TFoos>
class Baz {
public:
  template <typename T>
  auto store(T value) {
    return std::get<std::vector<T>>(data).push_back(value);
  }

private:
  std::tuple<std::vector<TFoos>...> data{};
};
```

Storing multiple types

```cpp
template <typename T, typename... Ts>
concept same_as_any = (... or std::same_as<T, Ts>);

template <CFoo... TFoos>
class Baz {
public:
  template <same_as_any<TFoos...> T>
  auto store(T value) {
    return std::get<std::vector<T>>(data).push_back(value);
  }

private:
  std::tuple<std::vector<TFoos>...> data{};
};
```

Storing multiple types

```cpp
// with virtual
Baz baz{};
baz.store(std::make_unique<Foo1>());
baz.store(std::make_unique<Foo2>());

// without virtual
Baz<Foo1, Foo2> baz{};
baz.store(Foo1{});
baz.store(Foo2{})
```

Storing multiple types

```
// with virtual
Baz baz{};
baz.store(std::make_unique<Foo1>());
baz.store(std::make_unique<Foo2>());

// without virtual
using foo_storage_t = Baz<Foo1, Foo2>;
foo_storage_t baz{};
baz.store(Foo1{});
baz.store(Foo2{})
```

Review

- Concepts bind interfaces
- Deduced class templates provide compile-time configurability of contained objects
  - Runtime configurability can be achieved with std::variant if absolutely needed
- Clever use of type lists and containers allows for statically typed storage of multiple types simultaneously – design will vary by use case

Downsides

- Increased translation unit size
- Potential increase to binary size
- May increase compile time
- May add complexity

A bold claim: As of C++ 20, for binaries built from source virtual is never required

Questions?

Practice time!

Task

- We want to monitor some set of devices on the same network that we are on
- Each device type is unique in how we must interact with it
- It is not possible to know the device's connection information before we join the network - we must find it in-situ

Design considerations

- Device detection
  - Easiest to find all devices of a single type at once
  - One scan per device type
- Device state monitoring
  - Need to allow each device type to have different communication mechanisms
  - Want to update state only on-command to avoid network overhead

Devices

```cpp
class DeviceInterface;
using device_list_t = std::vector<std::unique_ptr<DeviceInterface>>;

class DeviceInterface {
public:
  [[nodiscard]] static virtual auto find_in_env() -> device_list_t = 0;

  virtual auto update() -> void = 0;
};
```

Devices

```cpp
class DeviceInterface;
using device_list_t = std::vector<std::unique_ptr<DeviceInterface>>;

class DeviceInterface {
public:
  virtual auto update() -> void = 0;
};
```

Devices

```cpp
class Switch final : DeviceInterface {
public:
  [[nodiscard]] static auto find_in_env() -> device_list_t {
    // Some device finding logic
  }

  auto update() -> void override { /* Update is_on */ }

private:
  bool is_on{false};
};

class Dimmer final : DeviceInterface {
public:
  [[nodiscard]] static auto find_in_env() -> device_list_t {
    // Some device finding logic
  }

  auto update() -> void override { /* Update brightness */ }

private:
  uint_fast8_t brightness{0};
};
```

Device manager

```
using device_list_t = std::vector<std::unique_ptr<DeviceInterface>>;

class DeviceManager {
public:
  DeviceManager(device_list_t devices_)
      : devices{std::move(devices_)} {}

  auto update() -> void {
    for (auto &device : devices) {
      device->update();
    }
  }

private:
  device_list_t devices{};
};
```

Device manager

```cpp
using device_list_t = std::vector<std::unique_ptr<DeviceInterface>>;

class DeviceManager {
public:
  [[nodiscard]] static auto get_devices() -> device_list_t {
    device_list_t output{};

    { // Switch
      auto device_list = Switch::find_in_env();
      output.insert(std::end(output),
                    std::make_move_iterator(std::begin(device_list)),
                    std::make_move_iterator(std::end(device_list)));
    }
    { // Dimmer
      auto device_list = Switch::find_in_env();
      output.insert(std::end(output),
                    std::make_move_iterator(std::begin(device_list)),
                    std::make_move_iterator(std::end(device_list)));
    }

    return output;
  }
  ...
};
```

Device manager

```cpp
using device_list_t = std::vector<std::unique_ptr<DeviceInterface>>;

class DeviceManager {
public:
  [[nodiscard]] static auto get_devices() -> device_list_t {
    device_list_t output{};

    { // Switch
      auto device_list = Switch::find_in_env();
      output.insert(std::end(output),
                    std::make_move_iterator(std::begin(device_list)),
                    std::make_move_iterator(std::end(device_list)));
    }
    { // Dimmer
      auto device_list = Dimmer::find_in_env();
      output.insert(std::end(output),
                    std::make_move_iterator(std::begin(device_list)),
                    std::make_move_iterator(std::end(device_list)));
    }

    return output;
  }
  ...
};
```

Usage

```cpp
auto main() -> int {
  DeviceManager manager(DeviceManager::get_devices());
  manager.update();
}
```

Devices

```
template <typename T>
concept CDevice = requires(T device) {
  { device.update() } -> std::same_as<void>;
};
```

Devices

```
template <typename T>
concept CDevice = requires(T device) {
  { T::find_in_env() } -> std::same_as<std::vector<T>>;
  { device.update() } -> std::same_as<void>;
};
```

Devices

```cpp
class Switch {
public:
  [[nodiscard]] static auto find_in_env() -> std::vector<Switch> {
    // Some device finding logic
  }

  auto update() -> void { /* Update is_on */ }

private:
  bool is_on{false};
};

class Dimmer {
public:
  [[nodiscard]] static auto find_in_env() -> std::vector<Dimmer> {
    // Some device finding logic
  }

  auto update() -> void { /* Update the brightness */ }

private:
  uint_fast8_t brightness{0};
};
```

Device manager

```cpp
template <CDevice... TDevices>
class DeviceManager {
public:
  DeviceManager() {}

  auto update() -> void {
    std::apply(
        [this](auto &... device_lists) {
          (update_device(device_lists), ...);
        },
        devices);
  }

private:
  using device_list_t = std::tuple<std::vector<TDevices>...>;

  auto update_device(auto &device_list) -> void {
    for (auto &device : device_list) {
      device.update();
    }
  }

  device_list_t devices{};
};
```

Device manager

```
template <CDevice... TDevices>
class DeviceManager {
public:
  DeviceManager() : devices{get_devices()} {}

  ...
private:
  using device_list_t = std::tuple<std::vector<TDevices>...>;

  [[nodiscard]] static auto get_devices() -> device_list_t {
    return std::tuple{TDevices::find_in_env()...};
  }

  device_list_t devices{};
};
```

Usage

```cpp
auto main() -> int {
  DeviceManager<Switch, Dimmer> manager{};
  manager.update();
}
```

Questions?