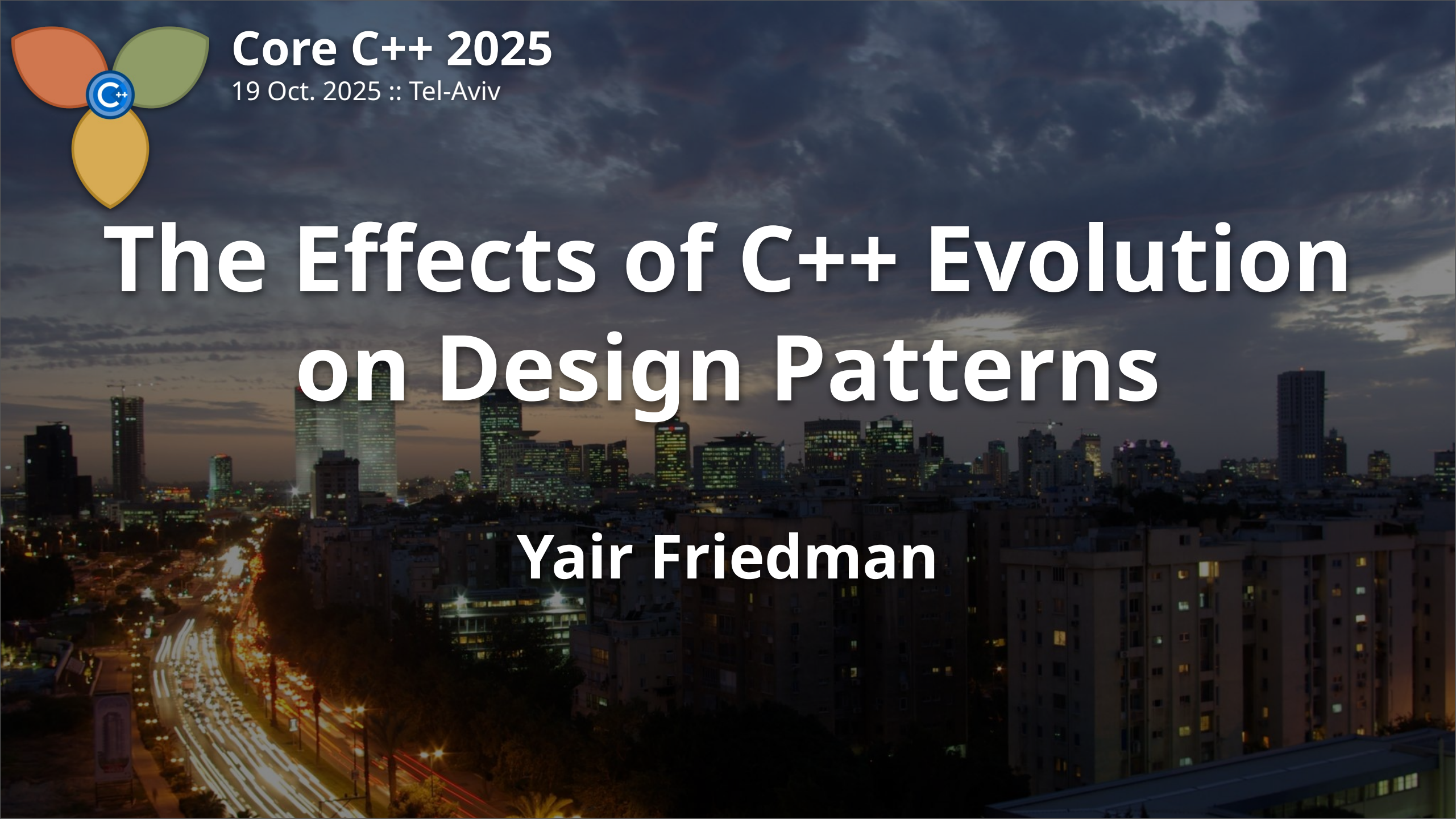

Core C++ 2025
19 Oct. 2025 :: Tel-Aviv
The Effects of C++ Evolution on Design Patterns
Yair Friedman

Who am I

- Enjoying coding in C++ for three decades
- Software Security Professional
- Co-author and instructor of several Modern C++ trainings
- Member of Israel National body for C++ standard

yair.f.lists@gmail.com

Agenda

- What are Design Patterns?
- What is Modren C++?
- The patterns themselves

Legend


```
int i = 5;
```

Regular "not that interesting" source code

```
using data_t = std::vector<int>;
```

Source code actually worth looking at ☺

```
int *a = nullptr;
```

Feature in subject

Design Patterns

What are Design Patterns?

- "Design pattern is a general, reusable solution to a commonly occurring problem in many contexts in software design"

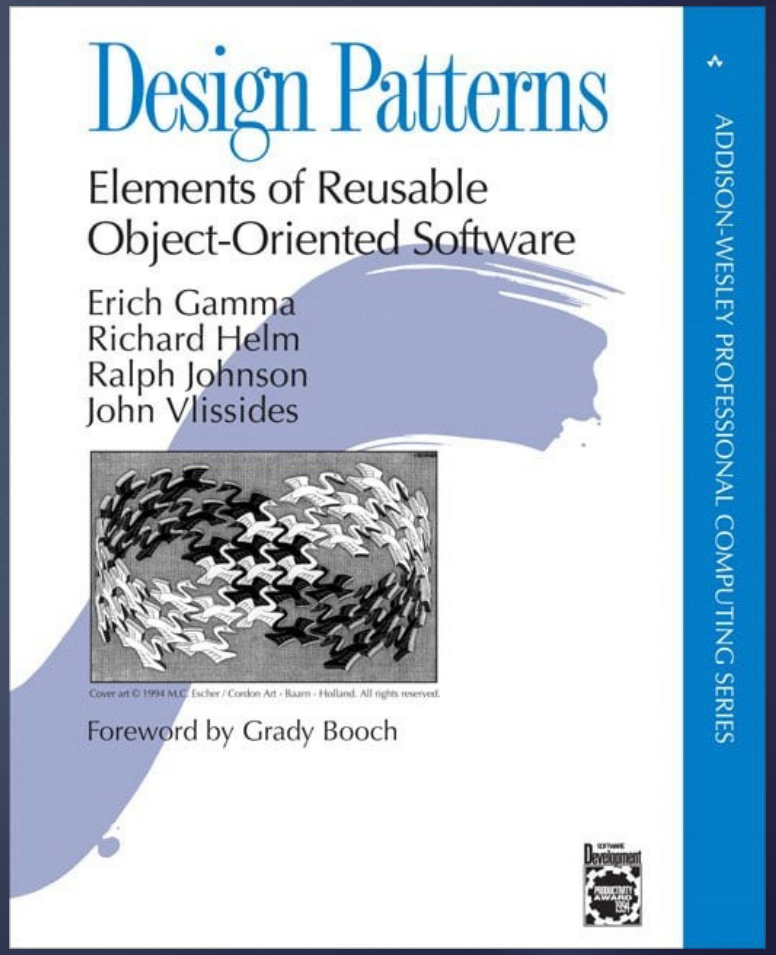
— Andrei Alexandrescu

- "The design patterns are descriptions of communicating **objects** and **classes** that are customized to solve a general design problem in a particular context."

— Gang of Four

Design Patterns: Elements of Reusable Object-Oriented Software (GoF)

- Published in 1994
- Strong Object-Oriented Design
- 23 classic software design patterns
- C++ code examples



Modern C++

What is Modren C++?

- “C++11 feels like a new language…”

— Bjarne Stroustrup

- “Modern C++ stands for C++, which is based on C++11, C++14, and C++17. ”
- “With C++11, we had a revolution. That revolution began with C++14 and become with C++17 to an evolution.”

— the late Rainer Grimm

The patterns themselves

The patterns themselves

- Patterns from GoF Book
- Eclectic list, ordered by “popularity”
- Code samples showing Modern C++ implementations


ABC

Abstract Base Class (1)

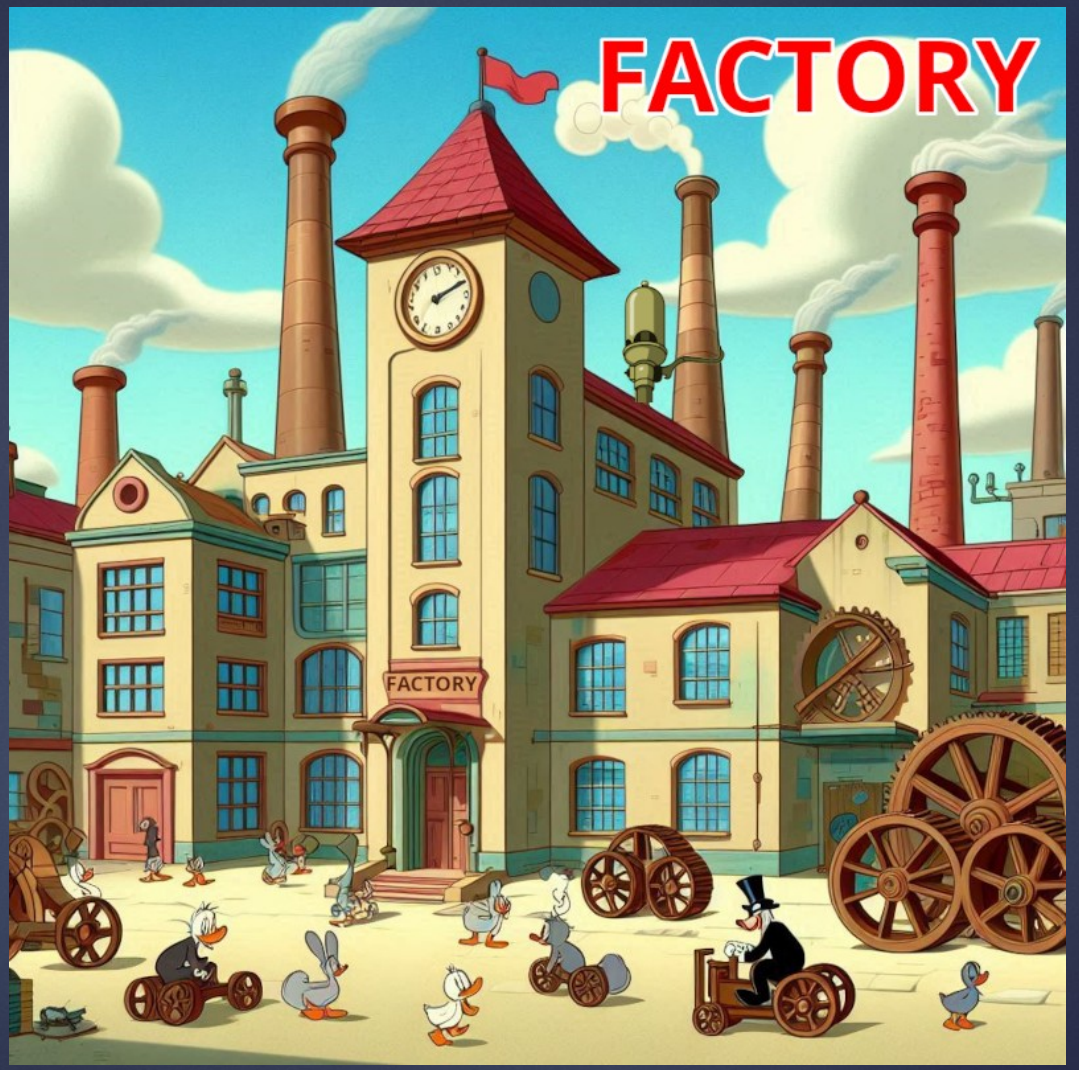

```
struct Abc {
  virtual ~Abc() = default;
  virtual void f() = 0;
  virtual void g() = 0;
};

struct Def : public Abc { // Default implementation class
  void f() override { std::cout << "Def::f\n"; }
  void g() override { std::cout << "Def::g\n"; }
protected:
  Def() = default; // prevents direct construction
};
```

Abstract Base Class (2)


```
struct G1 : public Def {
  void g() override { std::cout << "g1::g\n";}
};
struct G2 : public Def {…};
struct G3 : public Def {…};
int main() {
  using namespace std;
  unique_ptr<Abc> g1 = make_unique<G1>(); g1->f(); g1->g();
  unique_ptr<Abc> g2 = make_unique<G2>(); g2->f(); g2->g();
  unique_ptr<Abc> g3 = make_unique<G3>(); g3->f(); g3->g();
  // Def badDef1; // error: ‘constexpr Def::Def()’ is protected within this context
  // unique_ptr<Abc> badDef2 = make_unique<Def>(); // error: ‘constexpr Def::Def()’
                                                   // is protected within this context
  // unique_ptr<Abc> badG1 = g1; // error: use of deleted function ‘std::unique_ptr<_Tp,
                                 // _Dp>::unique_ptr(const std::unique_ptr<_Tp, _Dp>&)
  return 0;
}
```

Abstract Base Class — Takeaways

1. `override` specifier – ensures correct overriding of virtual functions.
2. `= default` for special member functions
3. `std::unique_ptr` – smart pointer for unique ownership and `std::make_unique` concise smart pointer creation
4. Movable (Non-copyable) objects


FACTORY
FACTORY

Factory

- "Creational Pattern" deals with object creation in a well-defined way
- Provide an interface for creating objects without specifying their concrete classes

Factory (1)


```
enum class AnimalSpecies { GenericAnimal, Dog, Cat, Rooster };
class Animal {
public:
  virtual ~Animal() = default;
  virtual void make_sound() const = 0;
  virtual void move() const = 0;
  virtual std::string get_name() const = 0;
};
class GenericAnimal : public Animal {
public:
  void make_sound() const override { std::cout << "a sound" << "\n"; }
  void move() const override { std::cout << "moves" << "\n"; };
  std::string get_name() const override { return "Animal"; };
};
```

Factory (2)

```
class Dog : public Animal {
public:
  void make_sound() const override { std::cout << "woof" << "\n"; }
  void move() const override {std::cout << "walks" << "\n";};
  std::string get_name() const override { return "Dog"; };
};

class Cat : public Animal {…};

class Rooster : public Animal {…};
};
```


Factory (3)


```
class AnimalFactory
{
  using Creator  = std::function<std::unique_ptr<Animal>()>;
  using Creators = std::unordered_map<AnimalSpecies, Creator>;
private:
  AnimalFactory() = default;
public:
  static std::unique_ptr<Animal> create_animal(AnimalSpecies animalSpecies) {
    static Creators creators = {
      {AnimalSpecies::Dog, []() {return std::make_unique<Dog>();} },
      {AnimalSpecies::Cat, []() {return std::make_unique<Cat>();} },
      {AnimalSpecies::Rooster, []() {return std::make_unique<Rooster>();} },
      {AnimalSpecies::GenericAnimal, []() {return std::make_unique<GenericAnimal>();} }
    };
    auto creator = creators.find(animalSpecies);
    if (creator == std::end(creators)) { // not found
      return std::make_unique<GenericAnimal>();
    }
    return creator->second();
  }
};
```

Factory (4)


```
int main() {
  std::vector<std::unique_ptr<Animal>> animals;
  animals.emplace_back(AnimalFactory::create_animal(AnimalSpecies::Dog));
  animals.emplace_back(AnimalFactory::create_animal(AnimalSpecies::Rooster));
  animals.emplace_back(AnimalFactory::create_animal(AnimalSpecies::Cat));
  animals.emplace_back(AnimalFactory::create_animal(AnimalSpecies::GenericAnimal));
  for (const auto& animal : animals) {
    std::cout << "The " << animal->get_name() << " says ";
    animal->make_sound();
    std::cout << "The " << animal->get_name() << ' ';
    animal->move();
  }
  return 0;
}
```

Factory — Takeaways

1. Use `auto` improves readability, maintainability, and type safety.
2. Use `std::function` and `std::unordered_map` to store the creators instead of `switch/if` chains
3. `using` Alias template instead of `typedef` to improve readability
4. `emplace_back` in place appending to `vector`
5. Range-based for-loops improved readability, safety, flexibility and efficiency
6. `enum class` enumerations for type safety and readability


SINGLETON

Singleton

- "Creational Pattern" deals with object creation in a well-defined way
- Ensure a class only has one instance, and provide a global point of access to it
- The most controversial Design Pattern — Seen by some as "anti-pattern"

Singleton (1)


```
class Singleton {
public:
  static Singleton& instance() {
    static Singleton singleton;
    return singleton;
  }
  std::string get_db_Connection() const { return db_connection_; }
private:
    Singleton() : db_connection_{"server=theserver;user=me"} {};
    ~Singleton() = default;
    std::string db_connection_;
};
void use_singleton(const std::string& name) {
    const auto& connection = Singleton::instance().get_db_Connection();
    std::cout << "singleton in use by " << name << " connection is " << connection
              << "object " << &connection << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(100)); // simulate some work
}
```

Singleton (2)


```
int main() {
  {
    std::thread thread1{&use_singleton, "thread1"};
    std::cout << "thread1 created " << thread1.get_id() << " " << std::endl;
    thread1.detach();
  }
  {
    std::thread thread2{&use_singleton, "thread2"};
    std::cout << "thread2 created " << thread2.get_id() << std::endl;
    thread2.detach();
  }
  using namespace std::chrono_literals;
  std::this_thread::sleep_for(2s); // give some time to thread's work
  std::cout << "Leaving main()" << std::endl;
}
```

Singleton — Takeaways

1. Use `static` function variable instead of raw pointers (Meyers Singleton)
2. Threads and thread safety — Threads didn't exist in the language
3. `chrono` time units and time literals

Templated Singleton (1)


```
Template<class T>
class Singleton {
public:
  static T& instance() {
    static T singleton{};
    return singleton;
  }
  // Singleton would not be copyable or movable
  Singleton(const Singleton&) = delete;
  Singleton& operator=(const Singleton&) = delete;
  Singleton(Singleton&&) noexcept = delete;
  Singleton& operator=(Singleton&&) noexcept = delete;
protected:
  Singleton() = default;
  virtual ~Singleton() = default;
};
```

Templated Singleton (2)


```
class A : public Singleton<A> {
public:
  friend class Singleton<A>;
  std::string getA() const { return a_resource_; }
  void doA(const std::string& name) const {
    std::cout << (std::stringstream{} << "singleton A in use by thread " << name
              << " the resource is " << getA()).str() << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(100)); // simulate some work
  }
protected:
    A() = default;  // call Singleton<A>'s protected default constructor
private:
  std::string a_resource_ = "The only A";
};
class B : public Singleton<B> {…};
void use_singleton(const std::string& name) {
  A::instance().doA(name);
  B::instance().doB(name);
}
int main() {...}
```

Templated Singleton — Takeaways

1. CRTP (static compile-time polymorphism) instead of runtime inheritance
2. Non-Movable objects


OBSERVER

Observer

- "Behavioral Pattern" identify common communication patterns among objects
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Observer (1)

```
class Observer {
public:
  virtual ~Observer() = default;
  virtual void update(const Subject& subject) = 0;
};
class Subject {
public:
  virtual ~Subject() = default;
  // The subscription management functions.
  virtual void attach(Observer& observer) = 0;
  virtual bool detach(Observer& observer) = 0;
  virtual void notify() const = 0;
};
```


Observer (2)


```
class DefaultSubject : public Subject {
  using ObserverRef = std::reference_wrapper<Observer>;
public:
  void attach(Observer& observer) override {
    observers_.emplace_back(observer);
  }
  bool detach(Observer& observer) override {
    const auto siz = observers_.size();
    observers_.erase(std::remove_if(begin(observers_), end(observers_),
                     [&observer](const auto& obs) {return &(obs.get()) == &observer;}),
                     end(observers_));
    return siz - observers_.size() > 0U;
  }
  void notify() const override {
    observers_count();
    for (const auto& observer : observers_) {observer.get().update(*this);}
  }
  void observers_count() const { … }
  static int next_seq() { … }
private:
  std::vector<ObserverRef> observers_;
};
```

Observer (3)


```
class MessageSubject : public DefaultSubject {
public:
  void create_message(const std::string& message = "Empty") {
    message_ = message;
    notify();
  }
  void some_business_logic() {
    message_ = "changing message to Don't Panic";
    notify();
    std::cout << "Subject about to do something\n";
  }
  std::string get_message() const {return message_;}
private:
  std::string message_;
};
```

Observer (4)


```
class MessageObserver : public Observer {
public:
  explicit MessageObserver(MessageSubject& subject) : subject_{subject}, id_{DefaultSubject::next_seq()} {
    subject_.attach(*this);
  }
  MessageObserver(const MessageObserver&) = delete; // (1)
  MessageObserver& operator=(const MessageObserver&) = delete; // (2)
  MessageObserver(MessageObserver&&) noexcept = delete; // (3)
  MessageObserver& operator=(MessageObserver&&) noexcept = delete; // (4)
  ~MessageObserver() { // (5)
    if (subject_.detach(*this)) {std::cout << "MessageObserver \"" << id_ << "\" detached from the list.\n";}
    else {std::cout << "MessageObserver \"" << id_ << "\" was already detached.\n";}
  }
  void update(const Subject& subject) override {
    if (&subject == &subject_) {std::cout << "MessageObserver \"" << id_ << "\": a new message is available: '"
                                          << subject_.get_message() << "'\n";}
  }
private:
  MessageSubject& subject_;
  int id_ = 0;
};
```

Observer (5)


```
int main() {
  auto subject = std::make_unique<MessageSubject>();
  auto observer1 = std::make_unique<MessageObserver>(*subject);
  auto observer2 = std::make_unique<MessageObserver>(*subject);
  subject->create_message("The Answer to the Ultimate Question of Life, "
                          "The Universe, and Everything is 42");
  observer2 = nullptr; // implicit detach
  subject->some_business_logic();
  subject->create_message("How many roads must a man walk down?");
  auto observer3  = std::make_unique<MessageObserver>(*subject);
  subject->detach(*observer1); // explicit detach
  observer1 = nullptr;         // still needs to nullify the pointer
  subject->create_message("What do you get if you multiply 6 by 9?");
  return 0;
}
```

Observer — Takeaways

1. `nullptr` is the NULL pointer, and use assignment to null as destructor invocation for smart pointers
2. `std::reference_wrapper` stores the observers as references in a vector, eliminating raw pointers in attach and detach. references are emplaced.
3. Using erase-remove idiom and lambda expressions for detach
4. `explicit` single-argument constructors
5. Non-Movable class prevents copying or moving

   Invoking 'Rule of five'
6. Unique pointers implementing RAII

Observer C++17 (1)


```
class Subject {
  using Callback = std::function<void(int, std::string_view)>;
  using IdCb = std::pair<int, Callback> ;
private:
  static int next_seq() { … }
  std::vector<IdCb> callbacks_;
public:
  int attach(Callback&& cb) {
    const auto id = next_seq();
    callbacks_.emplace_back(id, std::move(cb));
    return id;
  }
  bool detach(int id) {
    using std::cout; // also invoke ADL
    auto is_registered = [id](const IdCb& id_cb) {return id_cb.first == id;};
    const auto it = find_if(begin(callbacks_), end(callbacks_), is_registered);
    if (it == end(callbacks_)) {return false;} // notfound
    callbacks_.erase(it);
    cout << "Detached callback " << id << ".\n";
    return true;
  }
  ...
```

Observer C++17 (2)


```
void notify(std::string_view msg) const {
    callbacks_count();
    for (const auto& pair : callbacks_ ) { // we don't really mind what's
      // the callback name is, as long as it matches the types (Callback type)
      pair.second(pair.first, msg);
    }
  }
  void callbacks_count() const { … }
};
class Observer {
public:
  void update(int id, std::string_view msg) {
    std::cout << "Observer " << id
              << "::update() called with msg "<< msg << ".\n";
  }
};
```

Observer C++17 (3)


```
int main() {
  Subject subject;
  Observer obs1; const auto obs1_id = subject.attach(std::bind(
                 std::mem_fn(&Observer::update), &obs1, std::placeholders::_1,
                                                        std::placeholders::_2));
  Observer obs2; const auto obs2_id = … ;
  subject.notify("The Answer to the Ultimate Question of Life, "
                 "The Universe, and Everything is 42");
  subject.detach(obs2_id);
  subject.notify("How many roads must a man walk down?");
  Observer obs3; [[maybe_unused]] const auto obs3_id = … ;
  subject.detach(obs1_id);
  subject.notify("What do you get if you multiply 6 by 9?");
  return 0;
}
```

Observer C++17 — Takeaways

1. Use `std::function` to store Observers as callbacks (C++11)
2. `std::bind` and `std::mem_fn` are used as callback wrappers instead of concrete observer classes (C++11)
3. ADL (Argument-dependent lookup) makes some expressions clearer (Pre- C++11)
4. Use non-owning `string_view` for notify strings
5. `[[maybe_unused]]` attribute suppress warnings

Observer C++20 (1)


```
class Subject;
class Observer;
class Scheduler;
// Coroutine: Task and Scheduler
// Task represents a coroutine, Promise is coroutine communication channel
struct Task {
  struct promise_type;
  using handle_type = std::coroutine_handle<promise_type>;
  struct promise_type {
    Task get_return_object() {return Task{handle_type::from_promise(*this)};}
    std::suspend_always initial_suspend() { return {}; }
    std::suspend_always final_suspend() noexcept { return {}; }
    void return_void() {}
    void unhandled_exception() {std::terminate();}
  };
```

Observer C++20 (2)


```
  Task(handle_type h) : handle(h) {}
  ~Task() {if (handle) {handle.destroy();}}
  // Task is movable, but when moving handle is invalidated
  Task(const Task&) = delete;
  Task& operator=(const Task&) = delete;
  Task(Task&& other) noexcept : handle(other.handle) {
    other.handle = nullptr;
  }
  Task& operator=(Task&& other) noexcept {
    if (this != &other) { if (handle) { handle.destroy(); }
      handle = other.handle; other.handle = nullptr;
    }
    return *this;
  }
  handle_type handle;
};
```

Observer C++20 (3)


```
class Scheduler {
public:
  void add_task(Observer* observer, Task&& task) {
    tasks_[observer].emplace_back(std::move(task));
  }
  void resume_tasks_for(Observer* observer) {
    auto it = tasks_.find(observer);
    if (it == tasks_.end()) return;
    resume_tasks(it->second);
  }
  void resume_all() { for (auto& [_, task_list] : tasks_) { resume_tasks(task_list); }}
  bool has_pending_tasks_for(Observer* observer) const {
    if (tasks_.count(observer) > 0U) {
      return std::any_of(cbegin(tasks_.at(observer)), cend(tasks_.at(observer)),
                         [](const auto& task) { return !task.handle.done();});
    }
    return false;
  }
...
```

Observer C++20 (4)


```
 bool has_any_pending_tasks() const {
    return std::any_of(std::cbegin(tasks_), std::cend(tasks_),
                       [](const auto& pair) { return !pair.second.empty(); });
  }
private:
  static void resume_tasks(std::vector<Task>& task_list) {
    for (auto& task : task_list) {
      if (task.handle && !task.handle.done()) {
        task.handle.resume();
      }
    }
    std::erase_if(task_list, [](const Task& task) { return task.handle.done(); });
  }
  std::unordered_map<Observer*, std::vector<Task>> tasks_;
};
```

Observer C++20 (5)


```
// Coroutine: Awaitable defines the behavior of a co_await
struct Awaitable {
  Scheduler& scheduler;
  Observer* observer;

  bool await_ready() const noexcept { return false; }
  void await_suspend(std::coroutine_handle<> handle) const {
    scheduler.add_task(observer, Task{std::coroutine_handle<Task::promise_type>
                                      ::from_address(handle.address())});
  }
  void await_resume() const noexcept {}
};
```

Observer C++20 (6)

```
class Observer {
public:
  virtual ~Observer() = default;
  // Pass std::string by value to copy it
  //   into the coroutine frame.
  virtual Task update(const Subject& subject,
                      const std::string message) = 0;
  virtual bool has_pending_tasks() const = 0;
};
```


Observer C++20 (7)


```
class Subject {
public:
  explicit Subject(Scheduler& scheduler) : scheduler_{scheduler} {}
  void attach(std::shared_ptr<Observer> observer) {
    observers_.emplace_back(std::move(observer));
  }
  void detach(Observer* observer) {
    using namespace std::chrono_literals;
    while (observer->has_pending_tasks()) {
      scheduler_.resume_tasks_for(observer);
      std::this_thread::sleep_for(100ms); // Sleep to not busy-wait
    }
    std::erase_if(observers_, [&](const std::shared_ptr<Observer>& obs) {return obs.get() == observer;});
  }
  void notify(const std::string& message) {
    for (const auto& observer : observers_) {
      scheduler_.add_task(observer.get(), observer->update(*this, message));
    }
  }
  static int next_seq() {static atomic_int seq = 0; return ++seq;}
private:
  std::vector<std::shared_ptr<Observer>> observers_;
  Scheduler& scheduler_;
};
```

Observer C++20 (8)


```
class MessageObserver : public Observer {
public:
  MessageObserver(Subject& subject, Scheduler& scheduler)
    : name_{std::format("Observer {}", subject.next_seq())},
      subject_{subject}, scheduler_{scheduler} {}
  ~MessageObserver() { subject_.detach(this); }
  Task update(const Subject& subject, const std::string message) override {
    if (&subject != &subject_) {co_return;}
    std::cout << format("{} received: '{}'\n", name_, message);
    // Simulate async steps
    co_await Awaitable{scheduler_, this};
    co_await Awaitable{scheduler_, this};
    std::cout << format("  {}: Done processing: '{}'\n", name_, message);
    co_return;
  }
...
```

Observer C++20 (9)


```
  bool has_pending_tasks() const override {
    return scheduler_.has_pending_tasks_for(const_cast<MessageObserver*>(this));
  }
private:
  std::string name_;
  Subject& subject_;
  Scheduler& scheduler_;
};

void business_logic(Scheduler& scheduler) {
  using namespace std::chrono_literals;
  while (scheduler.has_any_pending_tasks()) {
    scheduler.resume_all();
    std::this_thread::sleep_for(50ms);
  }
}
```

Observer C++20 (10)


```
int main() {
  Scheduler scheduler;
  auto subject = make_shared<Subject>(scheduler);
  auto observer1 = make_shared<MessageObserver>(*subject, scheduler); subject->attach(observer1);
  auto observer2 = make_shared<MessageObserver>(*subject, scheduler); subject->attach(observer2);
  subject->notify("The Answer to the Ultimate Question of Life, "
                  "The Universe, and Everything is 42");
  subject->detach(observer3);
  business_logic(scheduler);
  subject->notify("How many roads must a man walk down?");
  auto observer3 = make_shared<MessageObserver>(*subject, scheduler); subject->attach(observer3);
  subject->detach(observer2);
  observer2 = nullptr;
  subject->notify("What do you get if you multiply 6 by 9?");
  return 0;
}
```

Observer C++20 — Takeaways

1. Coroutine processing the messages asynchronously
    - coroutine Task, Promise and Awaitable
    - coroutine Scheduler
    - coroutine keywords
        - `co_return`
        - `co_await`
2. `erase_if` replaces erase-remove
3. Structured binding for assigning multiple values returned from a function, struct or tuple (C++17)
4. `any_of` instead of loop search (C++11)
5. `std::shared_ptr` – smart pointer for shared ownership and `std::make_shared` concise smart pointer creation (C++11)
6. `std::format` for formatted string

Q & A