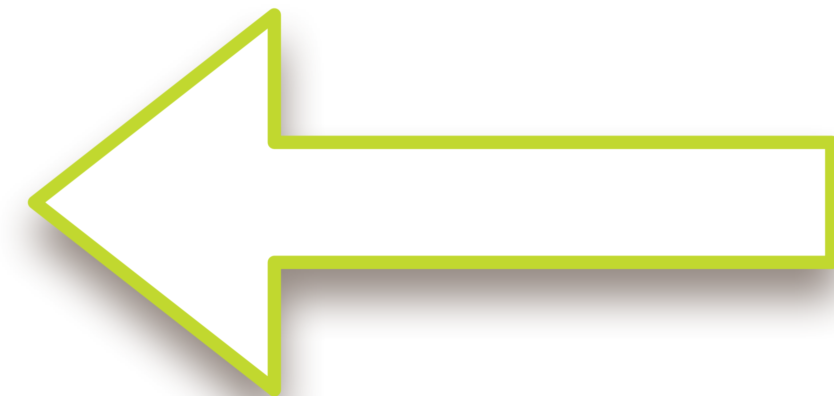
Polymorphic Task Template in Ten

Sean Parent

```cpp
template <class>
class task;

//...

int main() {
    task<unique_ptr<int>()> f = [_p = make_unique<int>(42)]() mutable {
        return move(_p);
    };

    cout << *f() << endl;
}
```

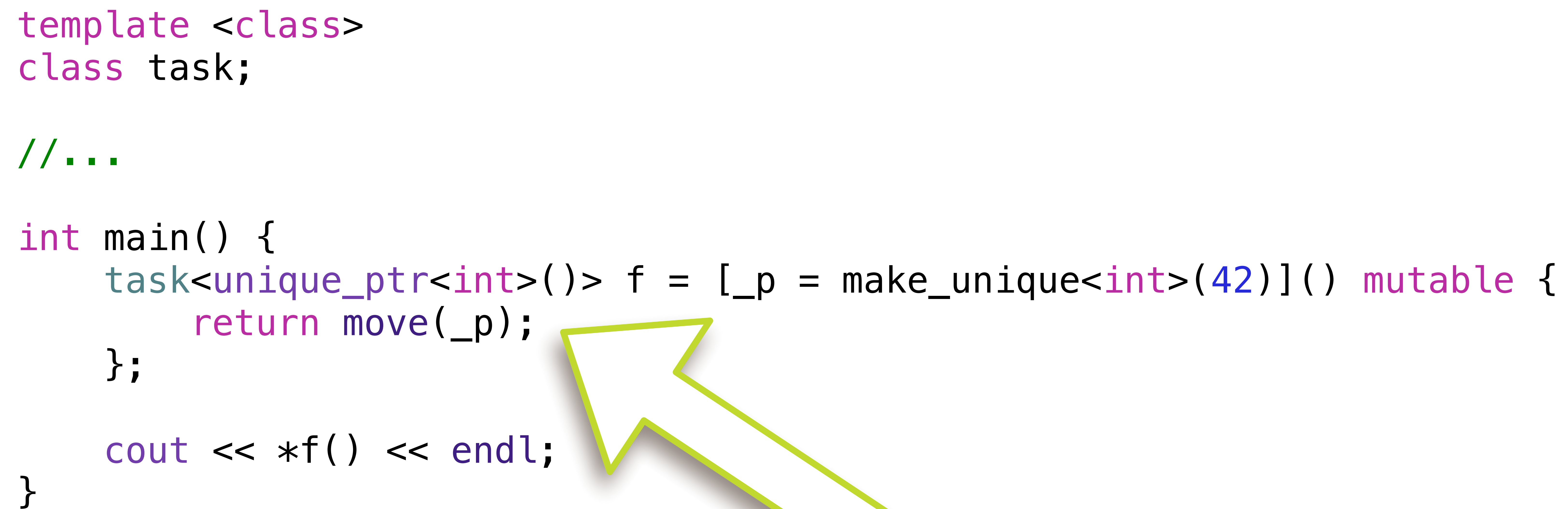
```cpp
template <class>
class task;

//...

int main() {
    task<unique_ptr<int>()> f = [_p = make_unique<int>(42)]() mutable {
        return move(_p);
    };

    cout << *f() << endl;
}
```



```cpp
template <class>
class task;

//...

int main() {
    task<unique_ptr<int>()> f = [_p = make_unique<int>(42)]() mutable {
        return move(_p);
    };

    cout << *f() << endl;
}
```

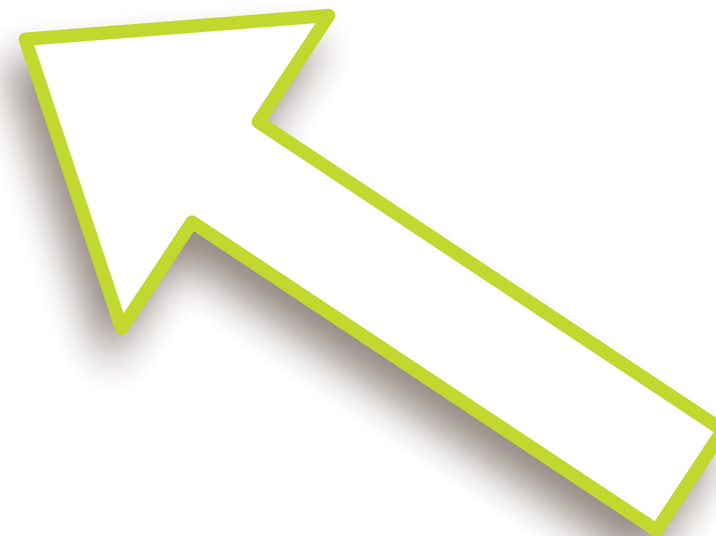
```cpp
template <class>
class task;

//...

int main() {
    task<unique_ptr<int>()> f = [_p = make_unique<int>(42)]() mutable {
        return move(_p);
    };

    cout << *f() << endl;
}
```

```
template <class>
class task;

//...

int main() {
    task<unique_ptr<int>()> f = [_p = make_unique<int>(42)]() mutable {
        return move(_p);
    };

    cout << *f() << endl;
}
```



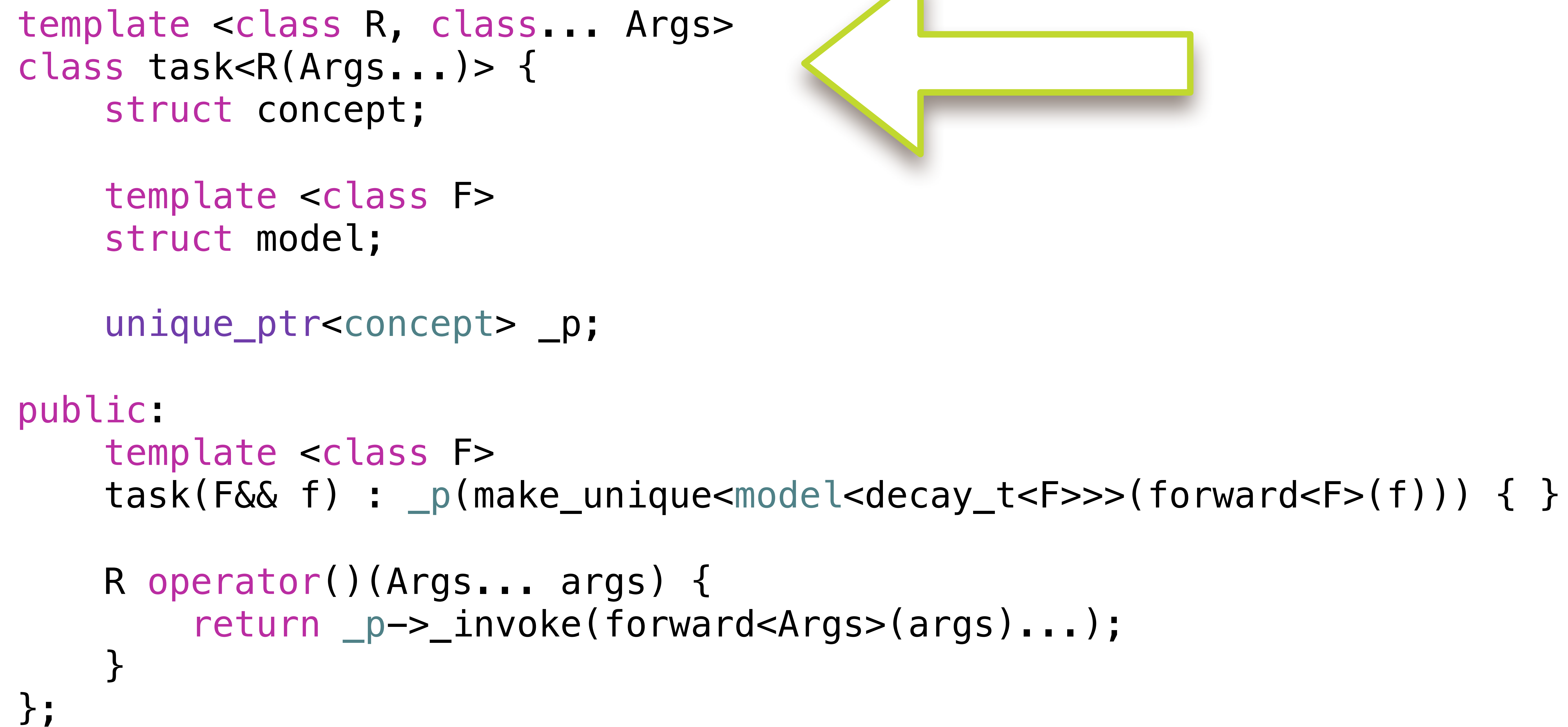
template <class>
class task;
//...
int main() {
task<unique_ptr<int>()> f = [_p = make_unique<int>(42)]() mutable {
return move(_p);
};
cout << *f() << endl;
}

```
template <class>
class task;

//...

int main() {
    task<unique_ptr<int>()> f = [_p = make_unique<int>(42)]() mutable {
        return move(_p);
    };

    cout << *f() << endl;
}
```

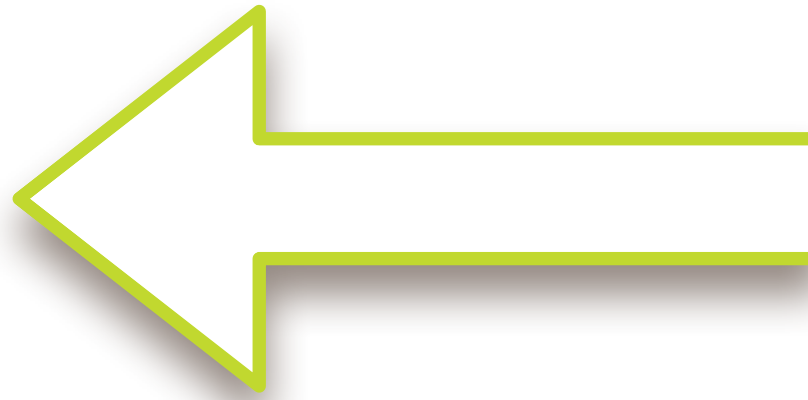


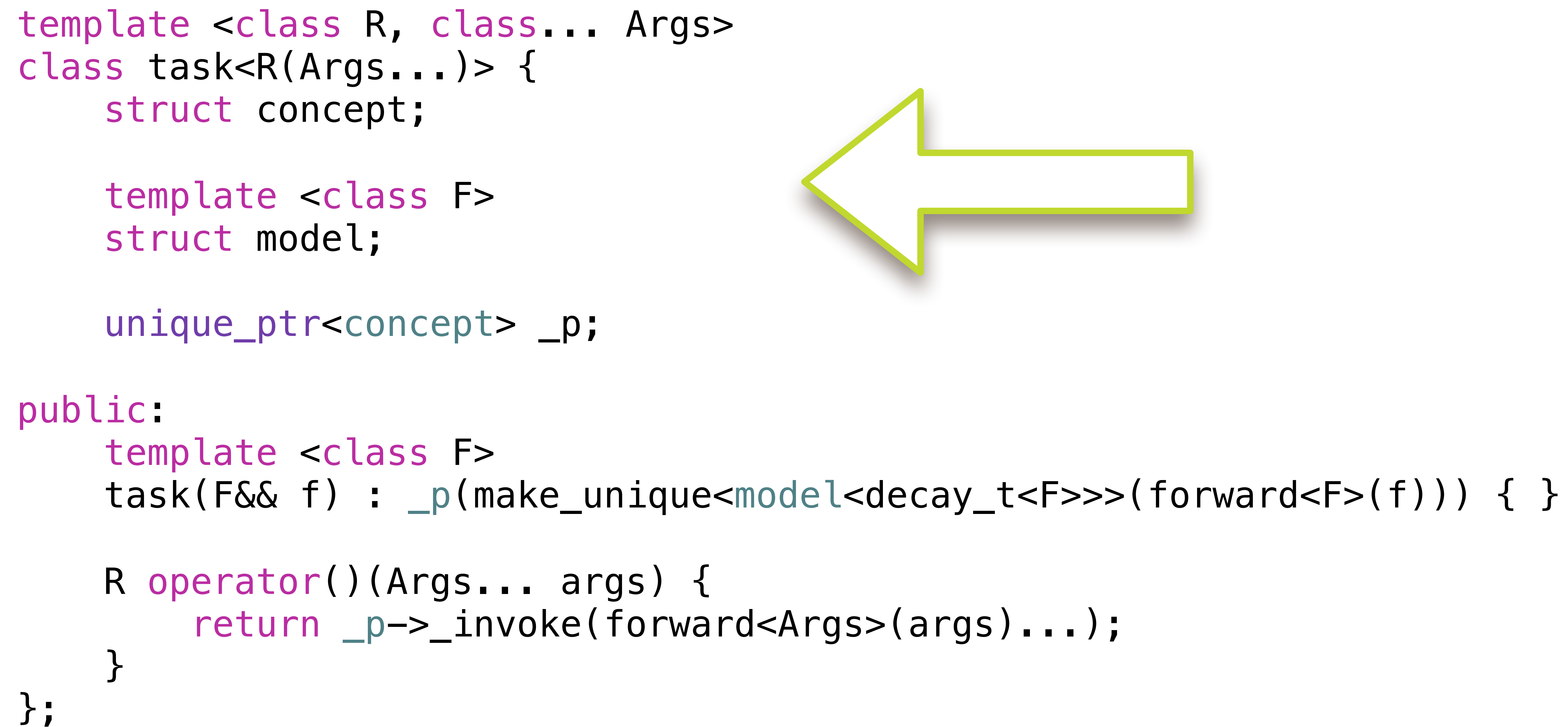
```
template <class>
class task;

//...

int main() {
    task<unique_ptr<int>()> f = [_p = make_unique<int>(42)]() mutable {
        return move(_p);
    };

    cout << *f() << endl;
}
```

42

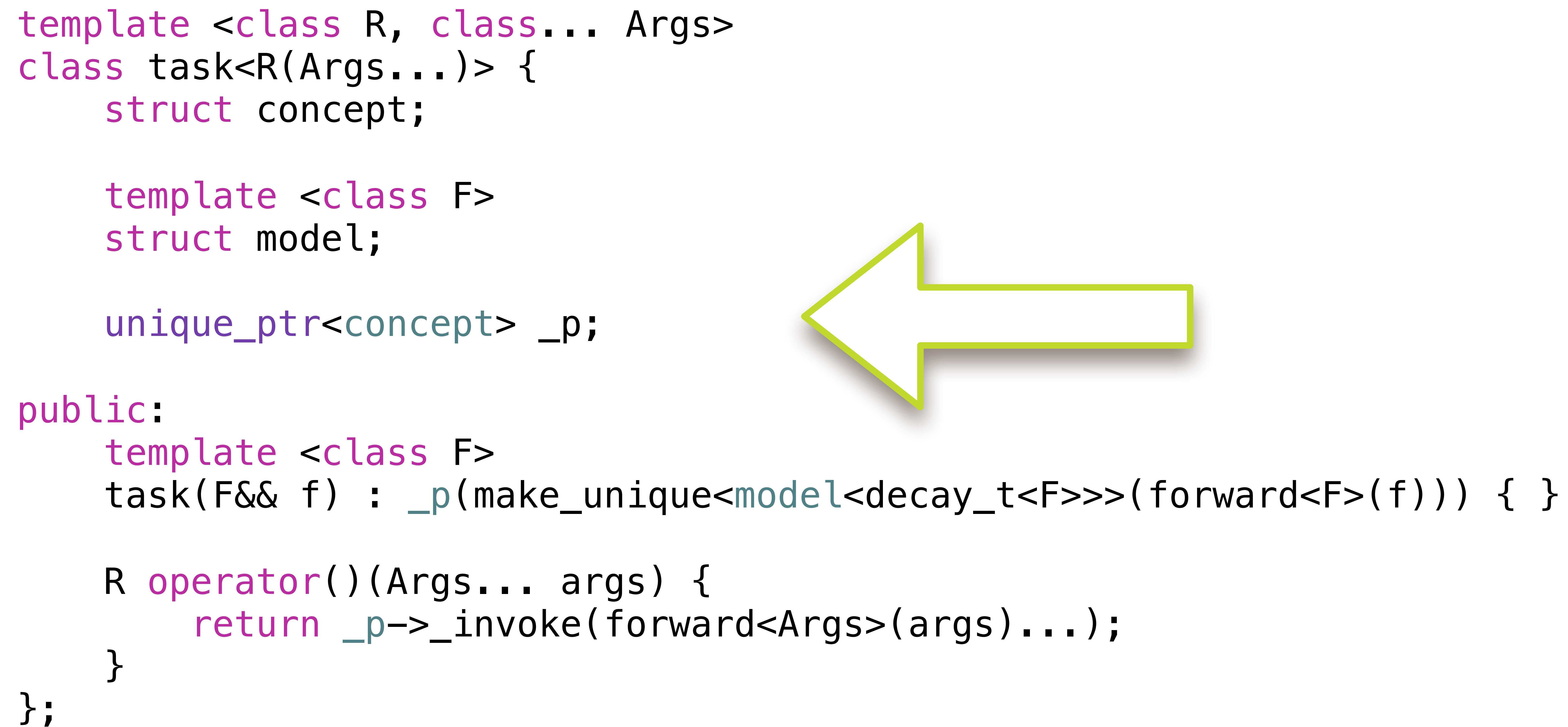

template <class R, class... Args>
class task<R(Args...)> {
struct concept;
template <class F>
struct model;
unique_ptr<concept> _p;
public:
template <class F>
task(F&& f) : _p(make_unique<model<decay_t<F>>>(forward<F>(f))) { }
R operator()(Args... args) {
return _p->_invoke(forward<Args>(args)...);
}
};

```
template <class R, class... Args>
class task<R(Args...)> {
    struct concept;

    template <class F>
    struct model;

    unique_ptr<concept> _p;

public:
    template <class F>
    task(F&& f) : _p(make_unique<model<decay_t<F>>>(forward<F>(f))) { }

    R operator()(Args... args) {
        return _p->_invoke(forward<Args>(args)...);
    }
};
```



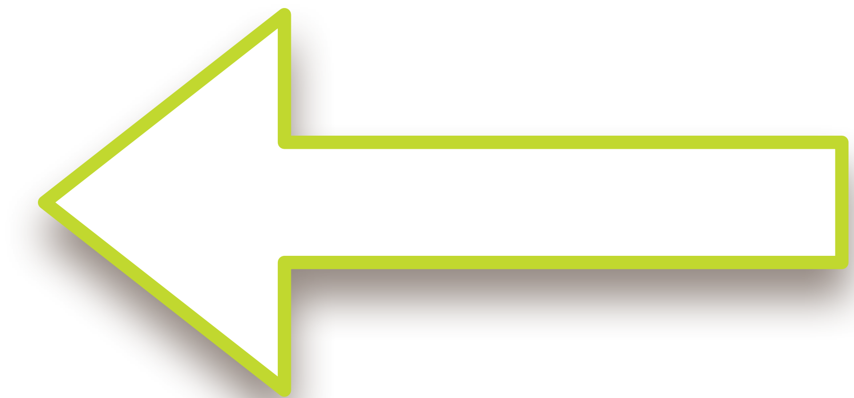


```
template <class R, class... Args>
class task<R(Args...)> {
    struct concept;

    template <class F>
    struct model;

    unique_ptr<concept> _p;

public:
    template <class F>
    task(F&& f) : _p(make_unique<model<decay_t<F>>>(forward<F>(f))) { }

    R operator()(Args... args) {
        return _p->_invoke(forward<Args>(args)...);
    }
};
```

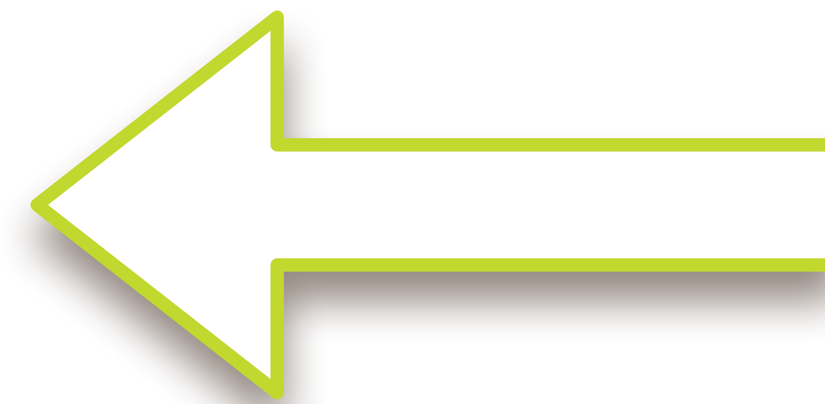


```
template <class R, class... Args>
class task<R(Args...)> {
    struct concept;

    template <class F>
    struct model;

    unique_ptr<concept> _p;

public:
    template <class F>
    task(F&& f) : _p(make_unique<model<decay_t<F>>>(forward<F>(f))) { }

    R operator()(Args... args) {
        return _p->_invoke(forward<Args>(args)...);
    }
};
```

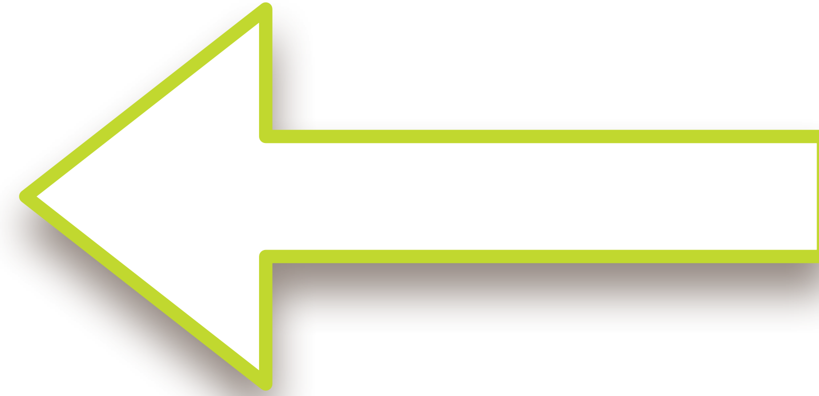
```cpp
template <class R, class... Args>
class task<R(Args...)> {
    struct concept;

    template <class F>
    struct model;

    unique_ptr<concept> _p;

public:
    template <class F>
    task(F&& f) : _p(make_unique<model<decay_t<F>>>(forward<F>(f))) { }

    R operator()(Args... args) {
        return _p->_invoke(forward<Args>(args)...);
    }
};
```



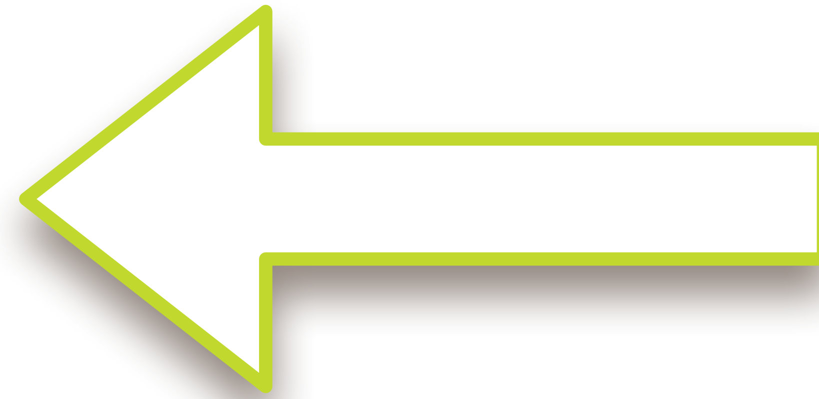
```cpp
template <class R, class... Args>
class task<R(Args...)> {
    struct concept;

    template <class F>
    struct model;

    unique_ptr<concept> _p;

public:
    template <class F>
    task(F&& f) : _p(make_unique<model<decay_t<F>>>(forward<F>(f))) { }

    R operator()(Args... args) {
        return _p->_invoke(forward<Args>(args)...);
    }
};
```



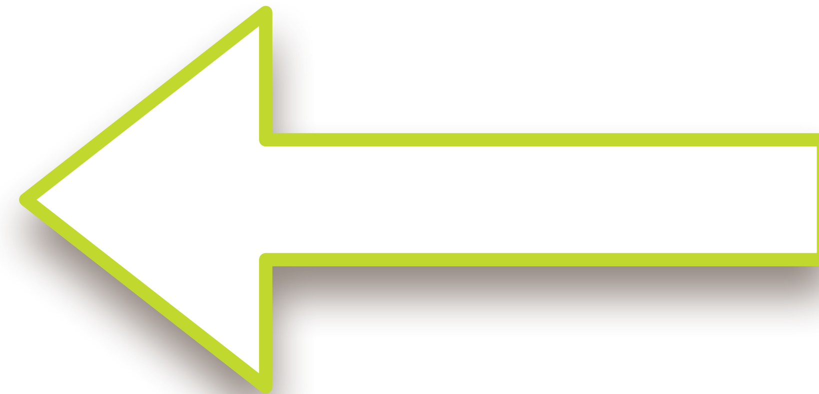
```cpp
template <class R, class... Args>
struct task<R(Args...)>::concept {

    virtual ~concept() = default;

    virtual R _invoke(Args&&...) = 0;

};
```



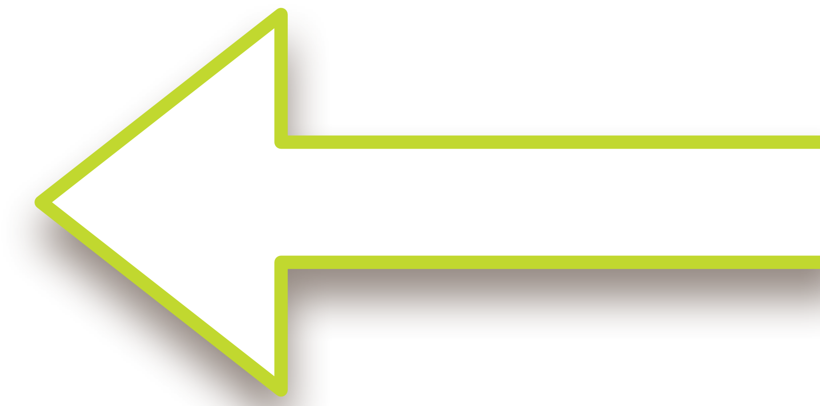
```
template <class R, class... Args>
struct task<R(Args...)>::concept {

    virtual ~concept() = default;

    virtual R _invoke(Args&&...) = 0;

};
```



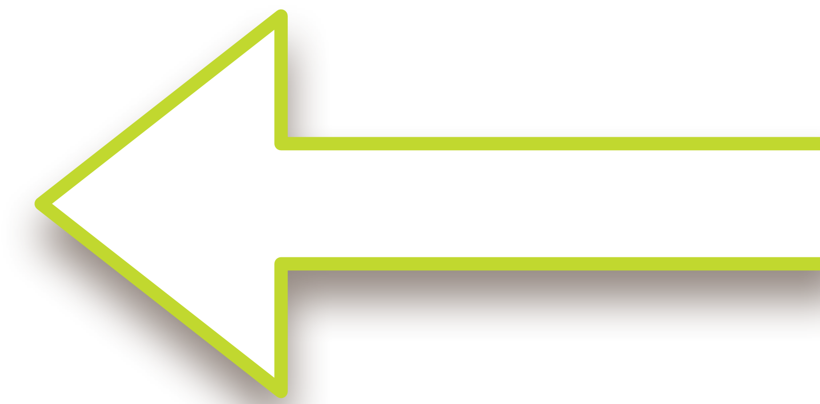
```
template <class R, class... Args>
struct task<R(Args...)>::concept {

    virtual ~concept() = default;

    virtual R _invoke(Args&&...) = 0;

};
```



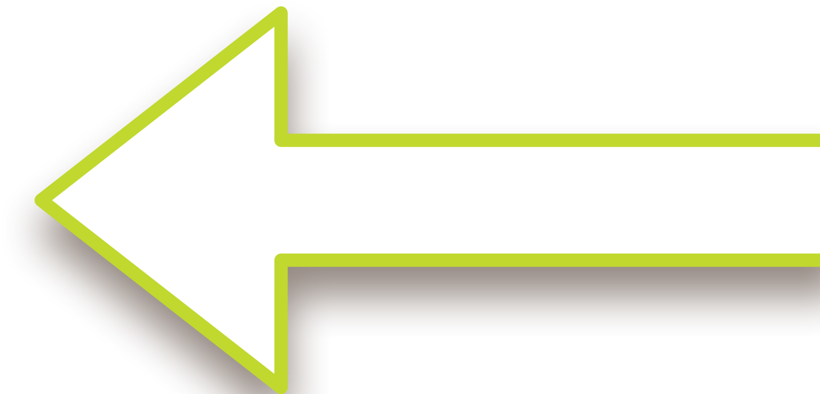
```cpp
template <class R, class... Args>
template <class F>
struct task<R(Args...)>::model final : concept {

    template <class G>
    model(G&& f) : _f(forward<G>(f)) { }

    R _invoke(Args&&... args) override {
        return invoke(_f, forward<Args>(args)…);
    }

    F _f;
};
```



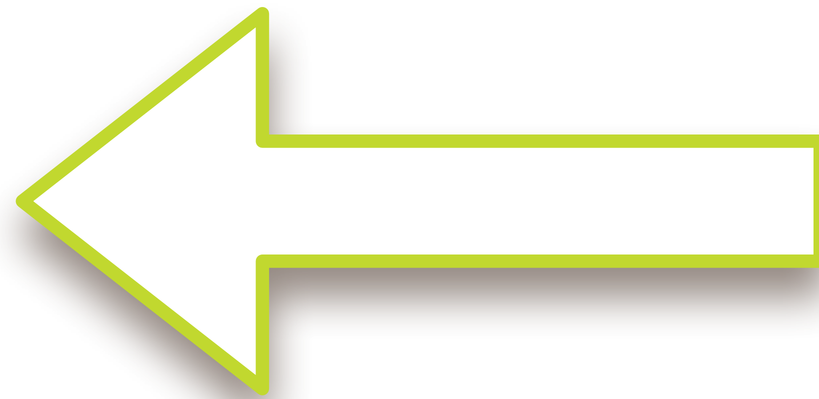
```cpp
template <class R, class... Args>
template <class F>
struct task<R(Args...)>::model final : concept {

    template <class G>
    model(G&& f) : _f(forward<G>(f)) { }

    R _invoke(Args&&... args) override {
        return invoke(_f, forward<Args>(args)…);
    }

    F _f;
};
```



```
template <class R, class... Args>
template <class F>
struct task<R(Args...)>::model final : concept {

    template <class G>
    model(G&& f) : _f(forward<G>(f)) { }

    R _invoke(Args&&... args) override {
        return invoke(_f, forward<Args>(args)…);
    }

    F _f;
};
```

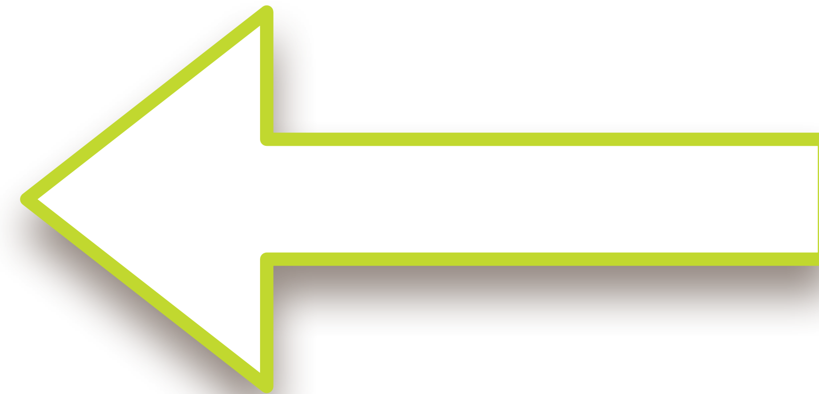


```
template <class R, class... Args>
template <class F>
struct task<R(Args...)>::model final : concept {

    template <class G>
    model(G&& f) : _f(forward<G>(f)) { }

    R _invoke(Args&&... args) override {
        return invoke(_f, forward<Args>(args)…);
    }

    F _f;
};
```



Done!

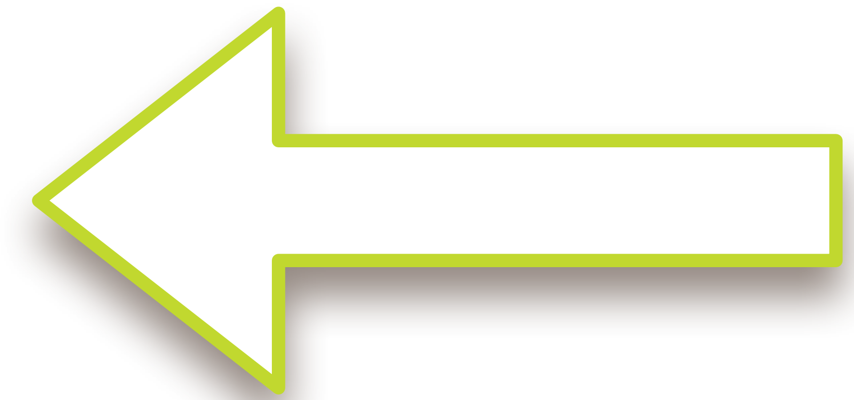
```
template <class R, class... Args>
class task<R(Args...)> {
    struct concept;

    template <class F, bool Small>
    struct model;

    static constexpr size_t small_size = sizeof(void*) * 4;

    aligned_storage_t<small_size> _data;

    concept& self() { return *static_cast<concept*>(static_cast<void*>(&_data)); }
public:
    template <class F>
    task(F&& f) {
        constexpr bool is_small = sizeof(model<decay_t<F>, true>) <= small_size;
        new (&_data) model<decay_t<F>, is_small>(forward<F>(f));
    }

    ~task() { self().~concept(); }

    task(task&& x) noexcept { x.self()._move(&_data); }
    task& operator=(task&& x) noexcept {
        self().~concept(); x.self()._move(&_data); return *this;
    }

    R operator()(Args... args) {
        return self()._invoke(forward<Args>(args)...);
    }
};
```



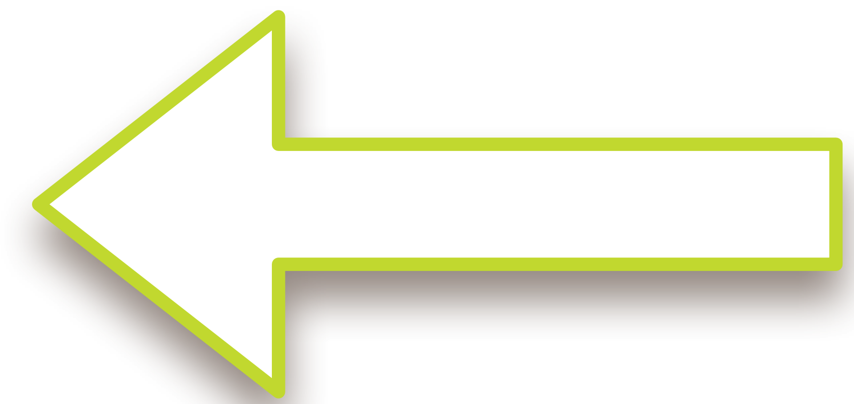
```
template <class R, class... Args>
class task<R(Args...)> {
    struct concept;

    template <class F, bool Small>
    struct model;

    static constexpr size_t small_size = sizeof(void*) * 4;

    aligned_storage_t<small_size> _data;

    concept& self() { return *static_cast<concept*>(static_cast<void*>(&_data)); }
public:
    template <class F>
    task(F&& f) {
        constexpr bool is_small = sizeof(model<decay_t<F>, true>) <= small_size;
        new (&_data) model<decay_t<F>, is_small>(forward<F>(f));
    }

    ~task() { self().~concept(); }

    task(task&& x) noexcept { x.self()._move(&_data); }
    task& operator=(task&& x) noexcept {
        self().~concept(); x.self()._move(&_data); return *this;
    }

    R operator()(Args... args) {
        return self()._invoke(forward<Args>(args)...);
    }
};
```

```
template <class R, class... Args>
class task<R(Args...)> {
    struct concept;

    template <class F, bool Small>
    struct model;

    static constexpr size_t small_size = sizeof(void*) * 4;

    aligned_storage_t<small_size> _data;

    concept& self() { return *static_cast<concept*>(static_cast<void*>(&_data)); }
public:
    template <class F>
    task(F&& f) {
        constexpr bool is_small = sizeof(model<decay_t<F>, true>) <= small_size;
        new (&_data) model<decay_t<F>, is_small>(forward<F>(f));
    }

    ~task() { self().~concept(); }

    task(task&& x) noexcept { x.self()._move(&_data); }
    task& operator=(task&& x) noexcept {
        self().~concept(); x.self()._move(&_data); return *this;
    }

    R operator()(Args... args) {
        return self()._invoke(forward<Args>(args)...);
    }
};
```

```
template <class R, class... Args>
class task<R(Args...)> {
    struct concept;

    template <class F, bool Small>
    struct model;

    static constexpr size_t small_size = sizeof(void*) * 4;

    aligned_storage_t<small_size> _data;

    concept& self() { return *static_cast<concept*>(static_cast<void*>(&_data)); }
public:
    template <class F>
    task(F&& f) {
        constexpr bool is_small = sizeof(model<decay_t<F>, true>) <= small_size;
        new (&_data) model<decay_t<F>, is_small>(forward<F>(f));
    }

    ~task() { self().~concept(); }

    task(task&& x) noexcept { x.self()._move(&_data); }
    task& operator=(task&& x) noexcept {
        self().~concept(); x.self()._move(&_data); return *this;
    }

    R operator()(Args... args) {
        return self()._invoke(forward<Args>(args)...);
    }
};
```



```
template <class R, class... Args>
class task<R(Args...)> {
    struct concept;

    template <class F, bool Small>
    struct model;

    static constexpr size_t small_size = sizeof(void*) * 4;

    aligned_storage_t<small_size> _data;

    concept& self() { return *static_cast<concept*>(static_cast<void*>(&_data)); }
public:
    template <class F>
    task(F&& f) {
        constexpr bool is_small = sizeof(model<decay_t<F>, true>) <= small_size;
        new (&_data) model<decay_t<F>, is_small>(forward<F>(f));
    }

    ~task() { self().~concept(); }

    task(task&& x) noexcept { x.self()._move(&_data); }
    task& operator=(task&& x) noexcept {
        self().~concept(); x.self()._move(&_data); return *this;
    }

    R operator()(Args... args) {
        return self()._invoke(forward<Args>(args)...);
    }
};
```



```
template <class R, class... Args>
class task<R(Args...)> {
    struct concept;

    template <class F, bool Small>
    struct model;

    static constexpr size_t small_size = sizeof(void*) * 4;

    aligned_storage_t<small_size> _data;

    concept& self() { return *static_cast<concept*>(static_cast<void*>(&_data)); }
public:
    template <class F>
    task(F&& f) {
        constexpr bool is_small = sizeof(model<decay_t<F>, true>) <= small_size;
        new (&_data) model<decay_t<F>, is_small>(forward<F>(f));
    }

    ~task() { self().~concept(); }

    task(task&& x) noexcept { x.self()._move(&_data); }
    task& operator=(task&& x) noexcept {
        self().~concept(); x.self()._move(&_data); return *this;
    }

    R operator()(Args... args) {
        return self()._invoke(forward<Args>(args)...);
    }
};
```

template <class R, class... Args>
class task<R(Args...)> {
    struct concept;

    template <class F, bool Small>
    struct model;

    static constexpr size_t small_size = sizeof(void*) * 4;

    aligned_storage_t<small_size> _data;

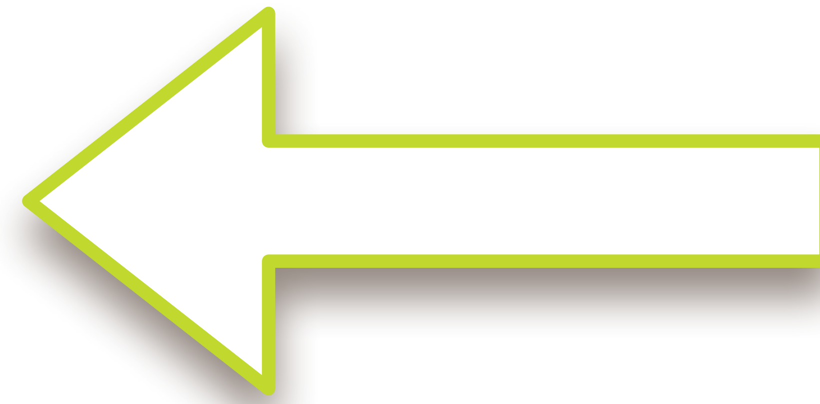
    concept& self() { return *static_cast<concept*>(static_cast<void*>(&_data)); }
public:
    template <class F>
    task(F&& f) {
        constexpr bool is_small = sizeof(model<decay_t<F>, true>) <= small_size;
        new (&_data) model<decay_t<F>, is_small>(forward<F>(f));
    }

    ~task() { self().~concept(); }

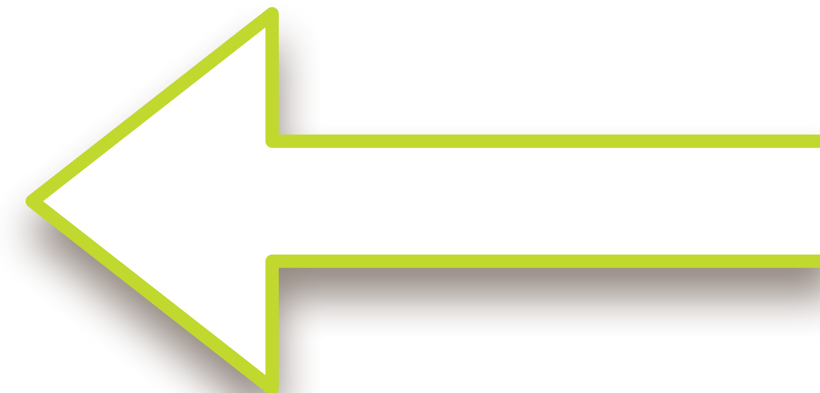
    task(task&& x) noexcept { x.self()._move(&_data); }
    task& operator=(task&& x) noexcept {
        self().~concept(); x.self()._move(&_data); return *this;
    }

    R operator()(Args... args) {
        return self()._invoke(forward<Args>(args)...);
    }
};

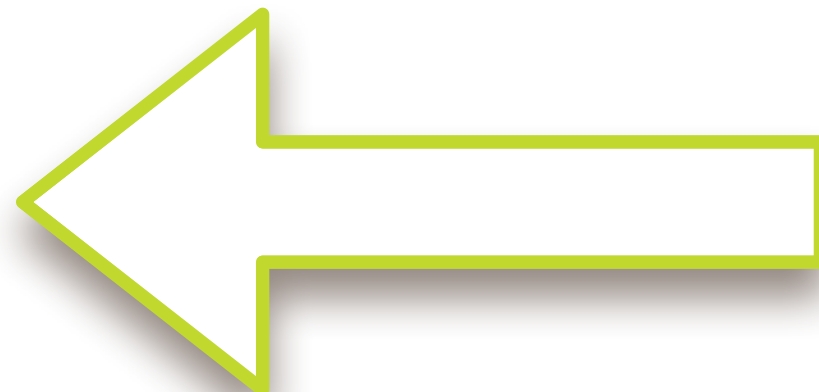
```
template <class R, class... Args>
class task<R(Args...)> {
    struct concept;

    template <class F, bool Small>
    struct model;

    static constexpr size_t small_size = sizeof(void*) * 4;

    aligned_storage_t<small_size> _data;

    concept& self() { return *static_cast<concept*>(static_cast<void*>(&_data)); }
public:
    template <class F>
    task(F&& f) {
        constexpr bool is_small = sizeof(model<decay_t<F>, true>) <= small_size;
        new (&_data) model<decay_t<F>, is_small>(forward<F>(f));
    }

    ~task() { self().~concept(); }

    task(task&& x) noexcept { x.self()._move(&_data); }
    task& operator=(task&& x) noexcept {
        self().~concept(); x.self()._move(&_data); return *this;
    }

    R operator()(Args... args) {
        return self()._invoke(forward<Args>(args)...);
    }
};
```

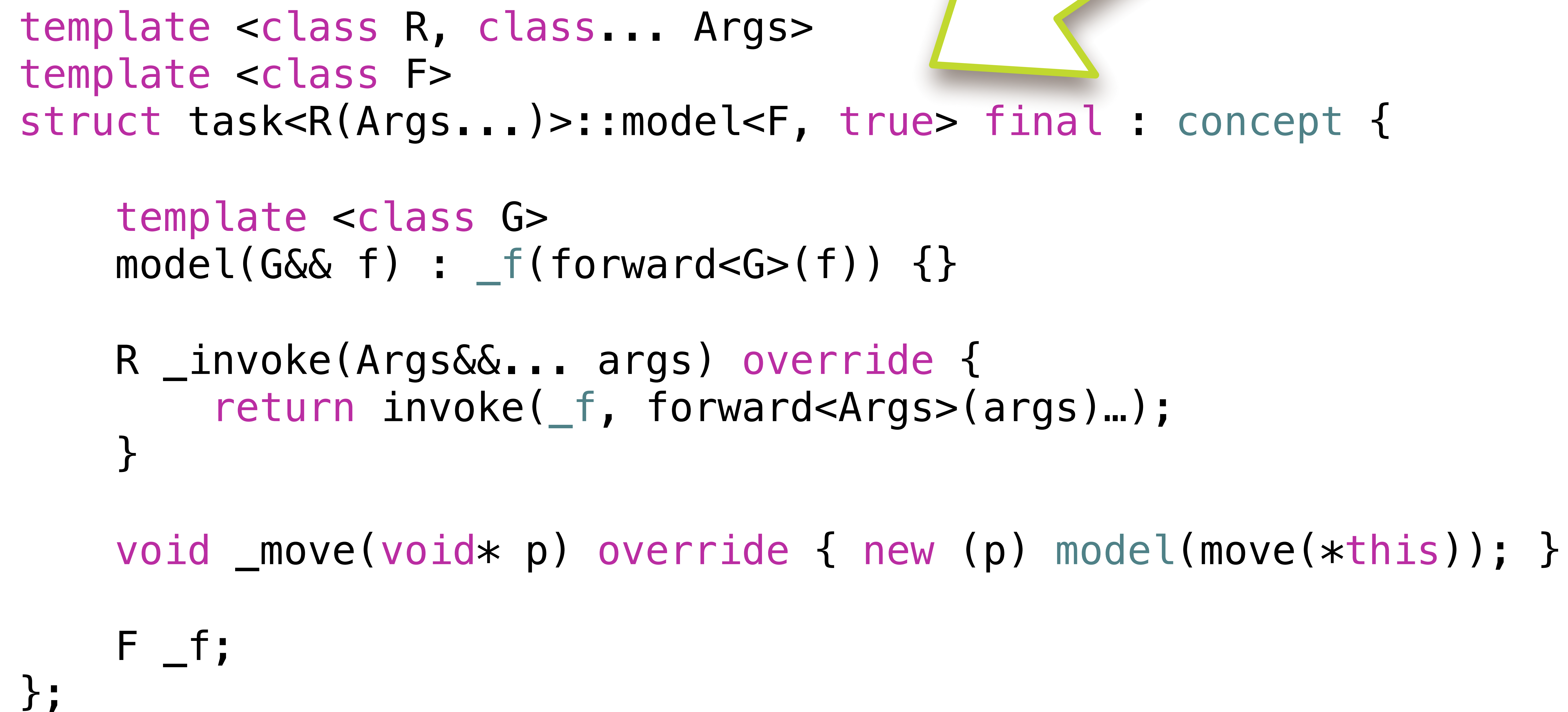


```cpp
template <class R, class... Args>
class task<R(Args...)> {
    struct concept;

    template <class F, bool Small>
    struct model;

    static constexpr size_t small_size = sizeof(void*) * 4;

    aligned_storage_t<small_size> _data;

    concept& self() { return *static_cast<concept*>(static_cast<void*>(&_data)); }
public:
    template <class F>
    task(F&& f) {
        constexpr bool is_small = sizeof(model<decay_t<F>, true>) <= small_size;
        new (&_data) model<decay_t<F>, is_small>(forward<F>(f));
    }

    ~task() { self().~concept(); }

    task(task&& x) noexcept { x.self()._move(&_data); }
    task& operator=(task&& x) noexcept {
        self().~concept(); x.self()._move(&_data); return *this;
    }

    R operator()(Args... args) {
        return self()._invoke(forward<Args>(args)...);
    }
};
```



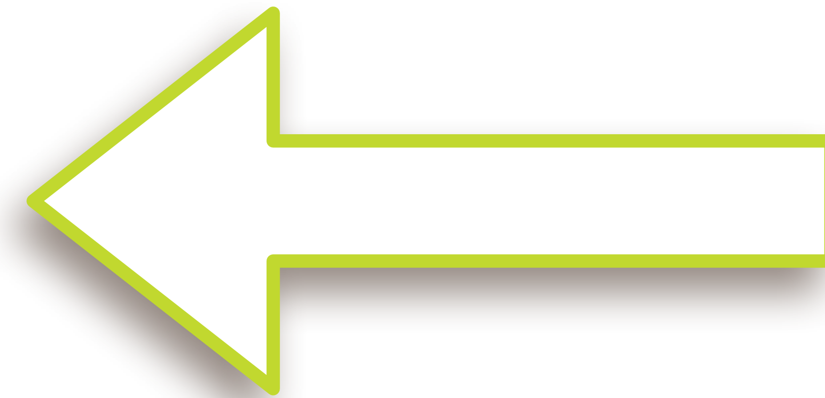
```
template <class R, class... Args>
struct task<R(Args...)>::concept {

    virtual ~concept() = default;

    virtual R _invoke(Args&&...) = 0;

    virtual void _move(void*) = 0;

};
```

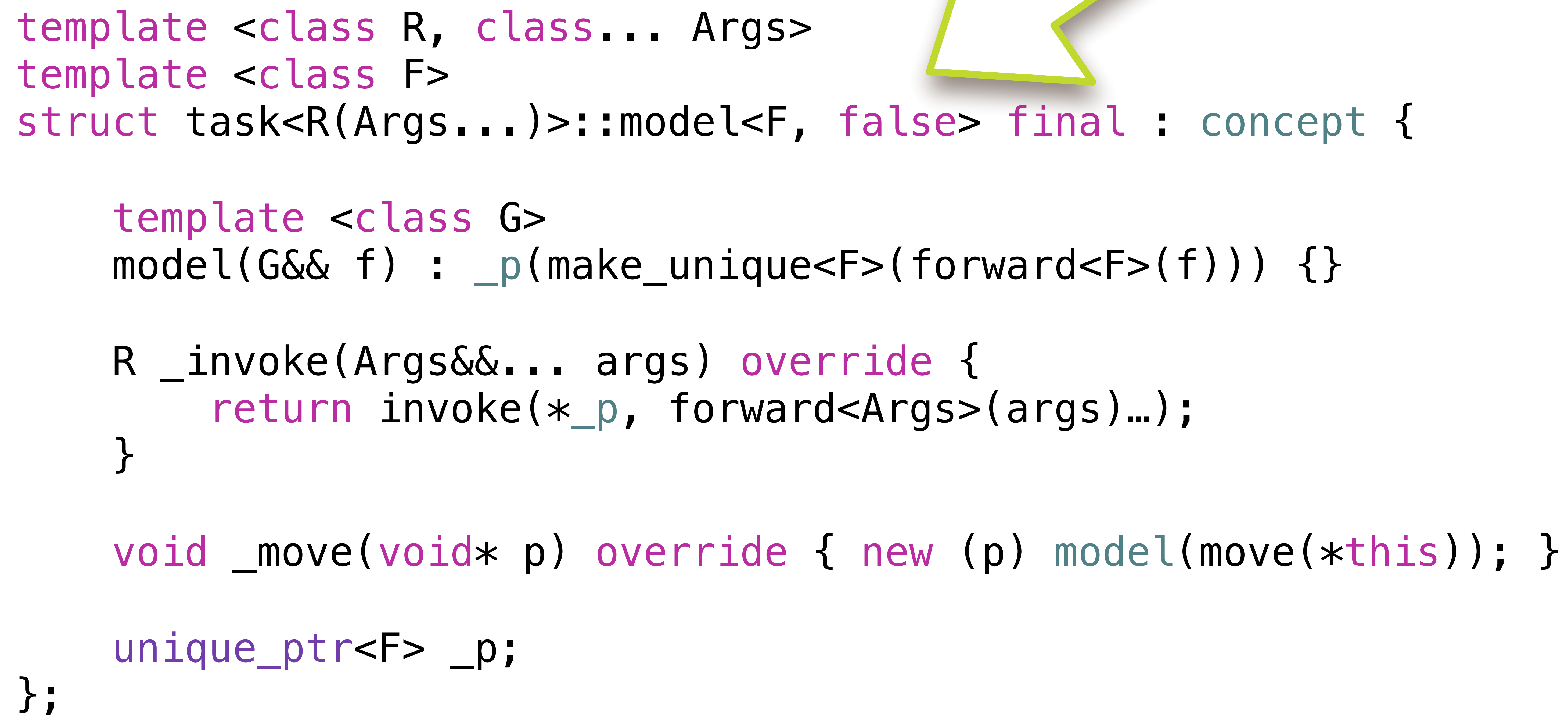




```
template <class R, class... Args>
template <class F>
struct task<R(Args...)>::model<F, true> final : concept {

    template <class G>
    model(G&& f) : _f(forward<G>(f)) {}

    R _invoke(Args&&... args) override {
        return invoke(_f, forward<Args>(args)…);
    }

    void _move(void* p) override { new (p) model(move(*this)); }

    F _f;
};
```

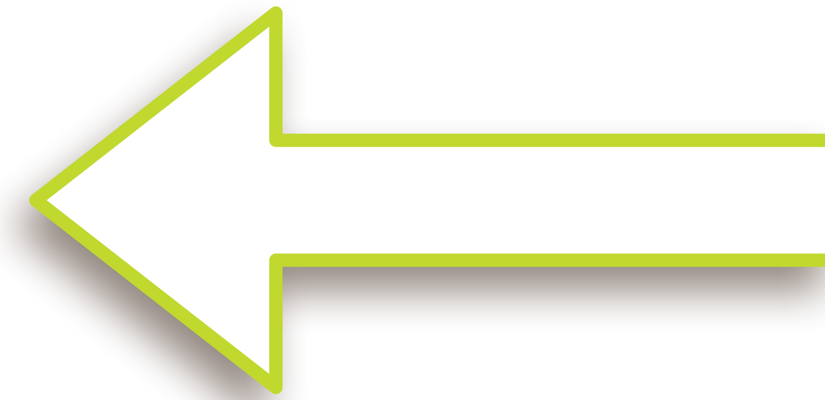
```cpp
template <class R, class... Args>
template <class F>
struct task<R(Args...)>::model<F, true> final : concept {

    template <class G>
    model(G&& f) : _f(forward<G>(f)) {}

    R _invoke(Args&&... args) override {
        return invoke(_f, forward<Args>(args)…);
    }

    void _move(void* p) override { new (p) model(move(*this)); }

    F _f;
};
```

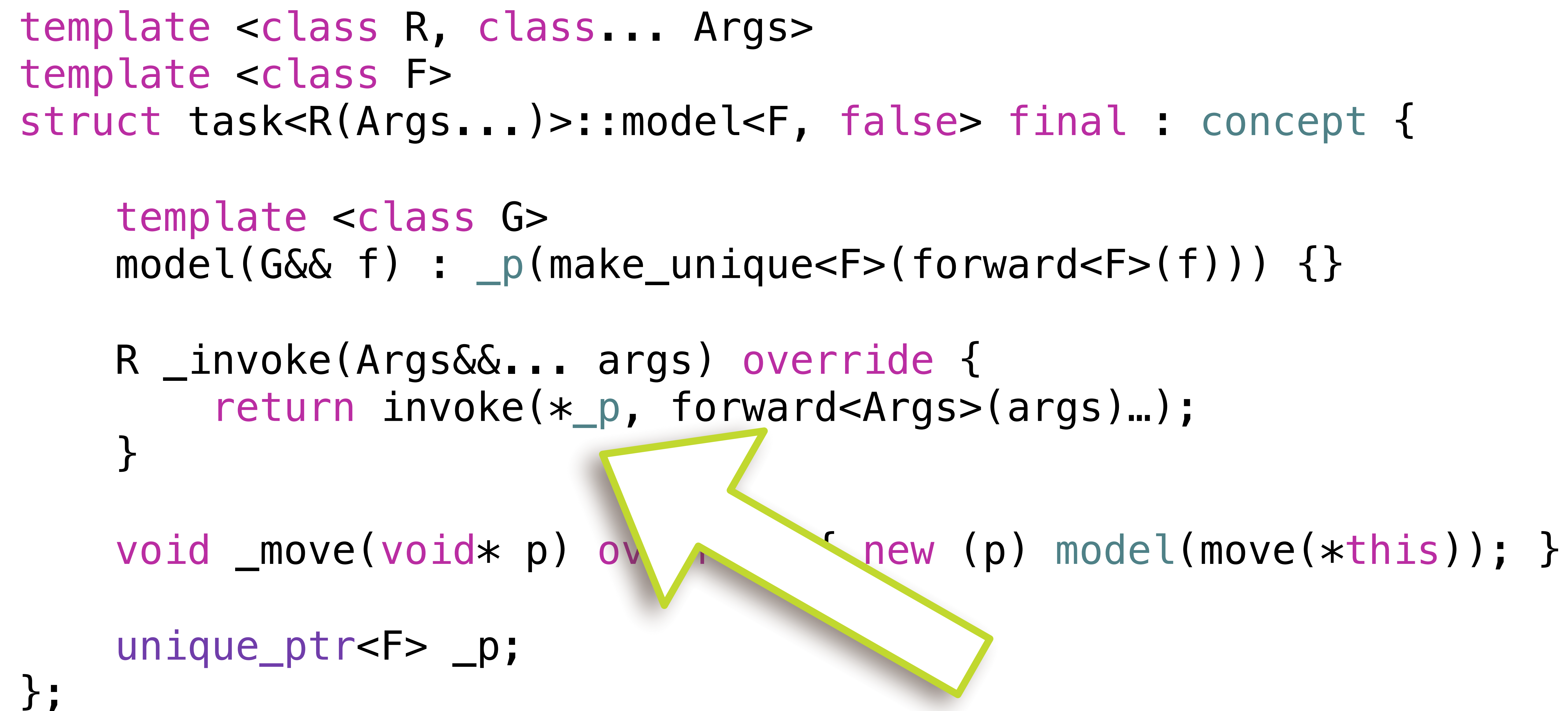




```
template <class R, class... Args>
template <class F>
struct task<R(Args...)>::model<F, false> final : concept {

    template <class G>
    model(G&& f) : _p(make_unique<F>(forward<F>(f))) {}

    R _invoke(Args&&... args) override {
        return invoke(*_p, forward<Args>(args)…);
    }

    void _move(void* p) override { new (p) model(move(*this)); }

    unique_ptr<F> _p;
};
```

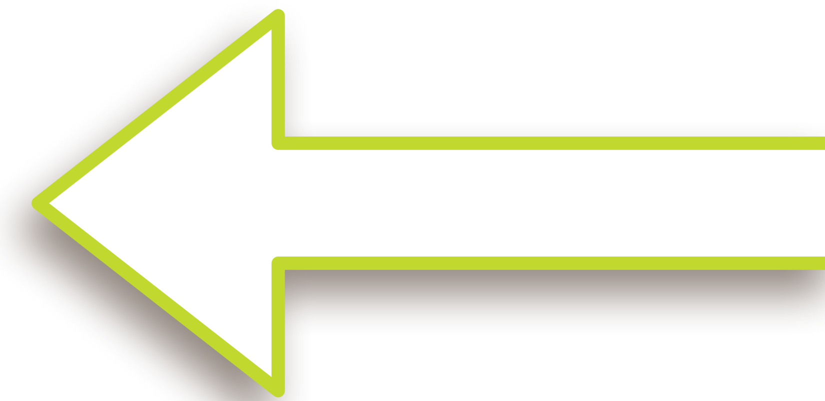
```
template <class R, class... Args>
template <class F>
struct task<R(Args...)>::model<F, false> final : concept {

    template <class G>
    model(G&& f) : _p(make_unique<F>(forward<F>(f))) {}

    R _invoke(Args&&... args) override {
        return invoke(*_p, forward<Args>(args)…);
    }

    void _move(void* p) override { new (p) model(move(*this)); }

    unique_ptr<F> _p;
};
```

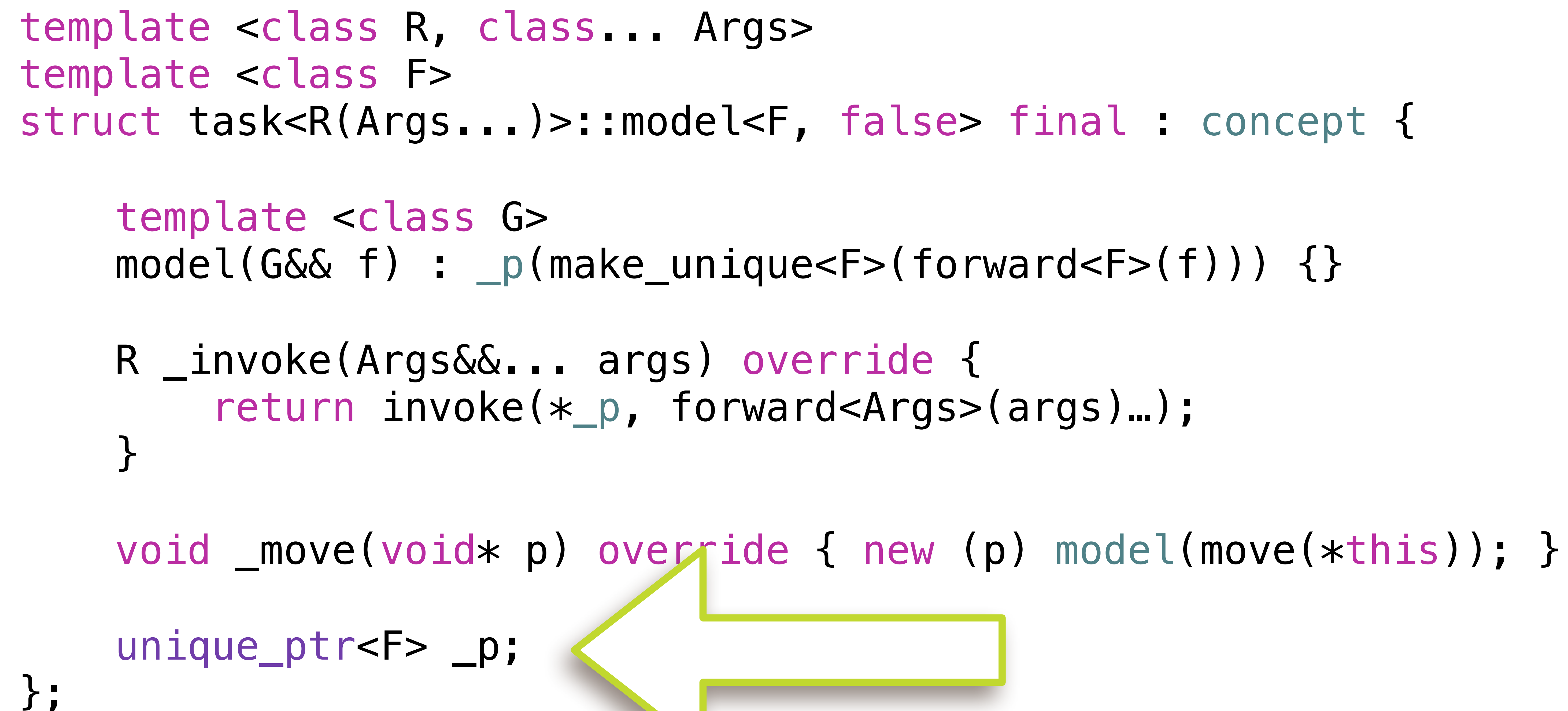




```
template <class R, class... Args>
template <class F>
struct task<R(Args...)>::model<F, false> final : concept {

    template <class G>
    model(G&& f) : _p(make_unique<F>(forward<F>(f))) {}

    R _invoke(Args&&... args) override {
        return invoke(*_p, forward<Args>(args)…);
    }

    void _move(void* p) ov[illegible] { new (p) model(move(*this)); }

    unique_ptr<F> _p;
};
```

```cpp
template <class R, class... Args>
template <class F>
struct task<R(Args...)>::model<F, false> final : concept {

    template <class G>
    model(G&& f) : _p(make_unique<F>(forward<F>(f))) {}

    R _invoke(Args&&... args) override {
        return invoke(*_p, forward<Args>(args)…);
    }

    void _move(void* p) override { new (p) model(move(*this)); }

    unique_ptr<F> _p;
};
```





```
template <class R, class... Args>
template <class F>
struct task<R(Args...)>::model<F, false> final : concept {

    template <class G>
    model(G&& f) : _p(make_unique<F>(forward<F>(f))) {}

    R _invoke(Args&&... args) override {
        return invoke(*_p, forward<Args>(args)…);
    }

    void _move(void* p) override { new (p) model(move(*this)); }

    unique_ptr<F> _p;
};
```

Polymorphic Task Template in Ten

Polymorphic Task Template in Ten

with Small Object Optimization

Done!

Done!

https://github.com/stlab/libraries/blob/develop/stlab/concurrency/task.hpp

Done!

https://github.com/stlab/libraries/blob/develop/stlab/concurrency/task.hpp

https://github.com/facebook/folly/blob/master/folly/docs/Poly.md