Beyond operator()

Zhihao Yuan

11/7/2022

How to make something callable?

Restrictions of operator()

Symantec

- You’re working on an object of a class type
- You own the class
- The action to “call” the object needs to have an unambiguous meaning

- Java ~~operator()~~
- C♯ ~~operator()~~
- Rust ~~operator()~~

Java

```java
interface Runnable {
    void run();
}
```

```java
class MyThread implements Runnable {
    public void run() {
        // code to execute
    }
}
```

```java
void addToPool(Runnable obj) { /* */ }
...

exec.addToPool(new MyThread());
```

Java with lambda

```
interface Runnable {
    void run();
}
```

```
void addToPool(Runnable obj) { /* */ }
…
exec.addToPool(() -> { /* code to execute */ });
```

Restrictions of operator()

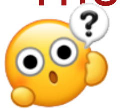
- You're working on an object of a class type
- You own the class
- ~~The action to "call" the object needs to have an unambiguous meaning~~

C#

Symantec

```
delegate int GetCount();
```

```
static void PrintCount(GetCount f)
{
    int i = f();
    System.Console.WriteLine("{0}", i);
}
```

```
string s = "The quick brown fox jumped over the lazy dog.";
PrintCount(
```


```
);
```

C# extension methods + delegates

```
public static class StringExtension
{
    public static int WordCount(this string str)
    {
        return /* impl code */;
    }
}

     string s = "The quick brown fox jumped over the lazy dog.";
     PrintCount(s.WordCount);
```

Restrictions of operator()

- You're working on an object of a class type
- ~~You own the class~~
- ~~The action to "call" the object needs to have an unambiguous meaning~~

Rust

```
trait FnOnce<Args> {
    type Output;
    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}
```

Rust

```rust
struct MyArray([i32; 3]);

impl FnOnce<(usize,)> for MyArray {
    type Output = i32;
    extern "rust-call" fn call_once(self, (i,): (usize,)) -> i32 {
        let MyArray(arr) = self;
        return arr[i];
    }
}
```

Restrictions of operator()

- ~~You're working on an object of a class type~~
- ~~You own the class~~
- ~~The action to "call" the object needs to have an unambiguous meaning~~

```
interface Runnable {
    void run();
}
```

This is a type-erasure

```
delegate int GetCount();
```

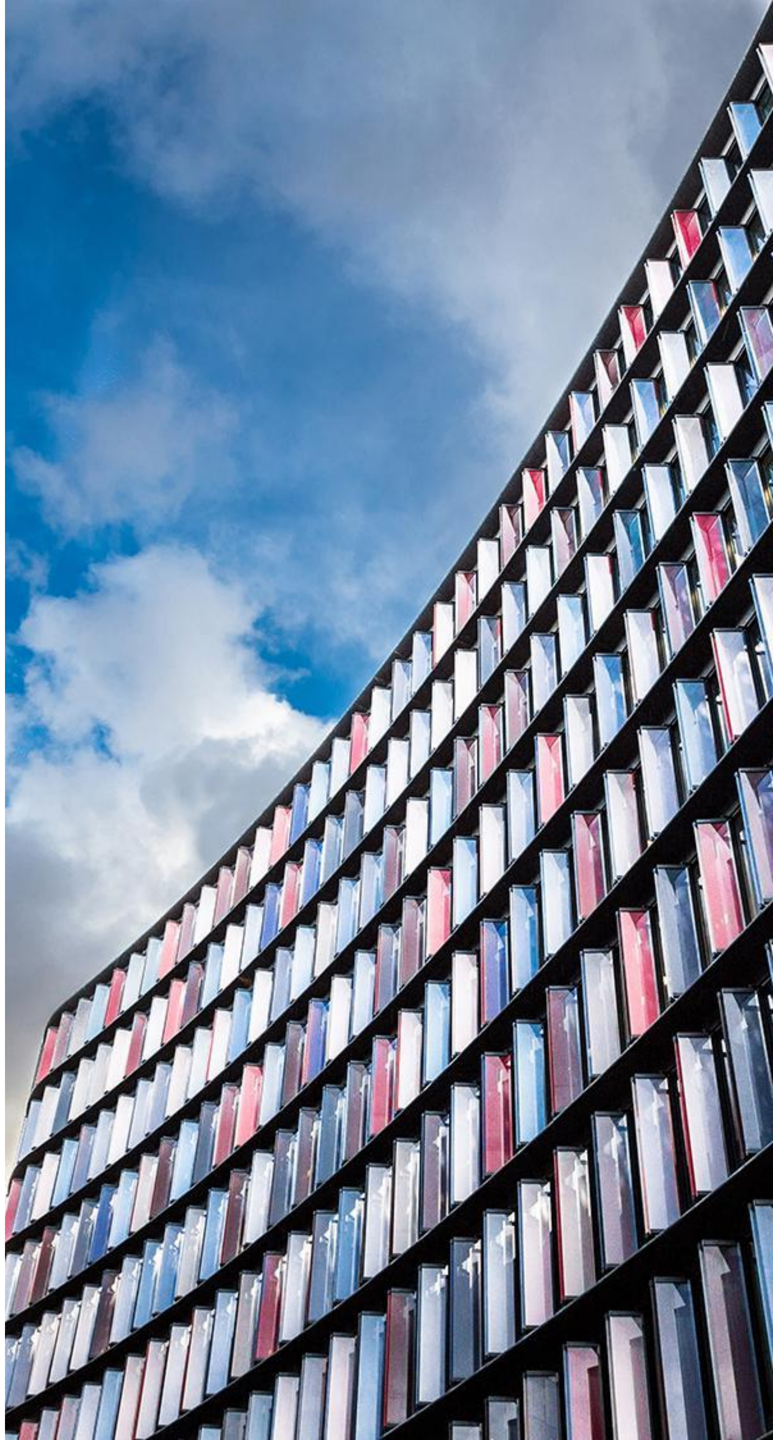
This is a type-erasure

```
fn foo(f: Box<dyn FnOnce(usize) -> i32>);
```

This is a type-erasure

What’s wrong with lambda, bind_front?

Symantec™



Example from the paper: lambda

```
pack.start([obj{std::move(obj)}]<class... T>(T &&...args) mutable
            { return obj.send(std::forward<T>(args)...); });
```

Example from the paper: bind_front

```
pack.start(std::bind_front(&Conn::send, std::move(obj)));
```

Example from the paper: proposed

```
pack.start({std::nontype<&Conn::send>, std::move(obj)});
```

Comments to address

Lambda can be made better

Bind_front can be made better

Constexpr argument can
replace the nontype tag

Lambda can be made better

```
// before
pack.start([obj{std::move(obj)}]<class... T>(T &&...args) mutable
            { return obj.send(std::forward<T>(args)...); });

// suggested (handle the case where 'send' is an overload set)
pack.start([obj{std::move(obj)}](auto &&...args)
            { return obj.send(>> args...); });
```

Response

Symantec

- Bind_front stills has cleaner semantics compared to lambda in this use case
- Still complicates stacks in debug information
- Moved twice (the original comment assumed &obj)
- Overload set can be handled in a better way when evolving the language:

Discussed in the paper

```
// proposed (doesn't handle overload set)
pack.start({std::nontype<&Conn::send>, std::move(obj)});

// proposed + vector-of-bool's "expression lambda"
pack.start({std::nontype<[][&1.send]>, std::move(obj)});

// proposed + p0834
pack.start({std::nontype<[].send>, std::move(obj)});
```

Bind_front can be made better

```
// before
pack.start(std::bind_front(&Conn::send, std::move(obj)));

// suggested
pack.start(std::bind_front(std::nontype<&Conn::send>, std::move(obj)));
```

Response

- Mental model is restricted to ‘bind’
- Still complicates stacks in debug information
- Moved twice
- nontype isn’t meant to be callable (more on that later)

Suggested in the paper

```
// before
pack.start(std::bind_front(&Conn::send, std::move(obj)));

// suggested
pack.start(std::bind_front<&Conn::send>(std::move(obj)));
```

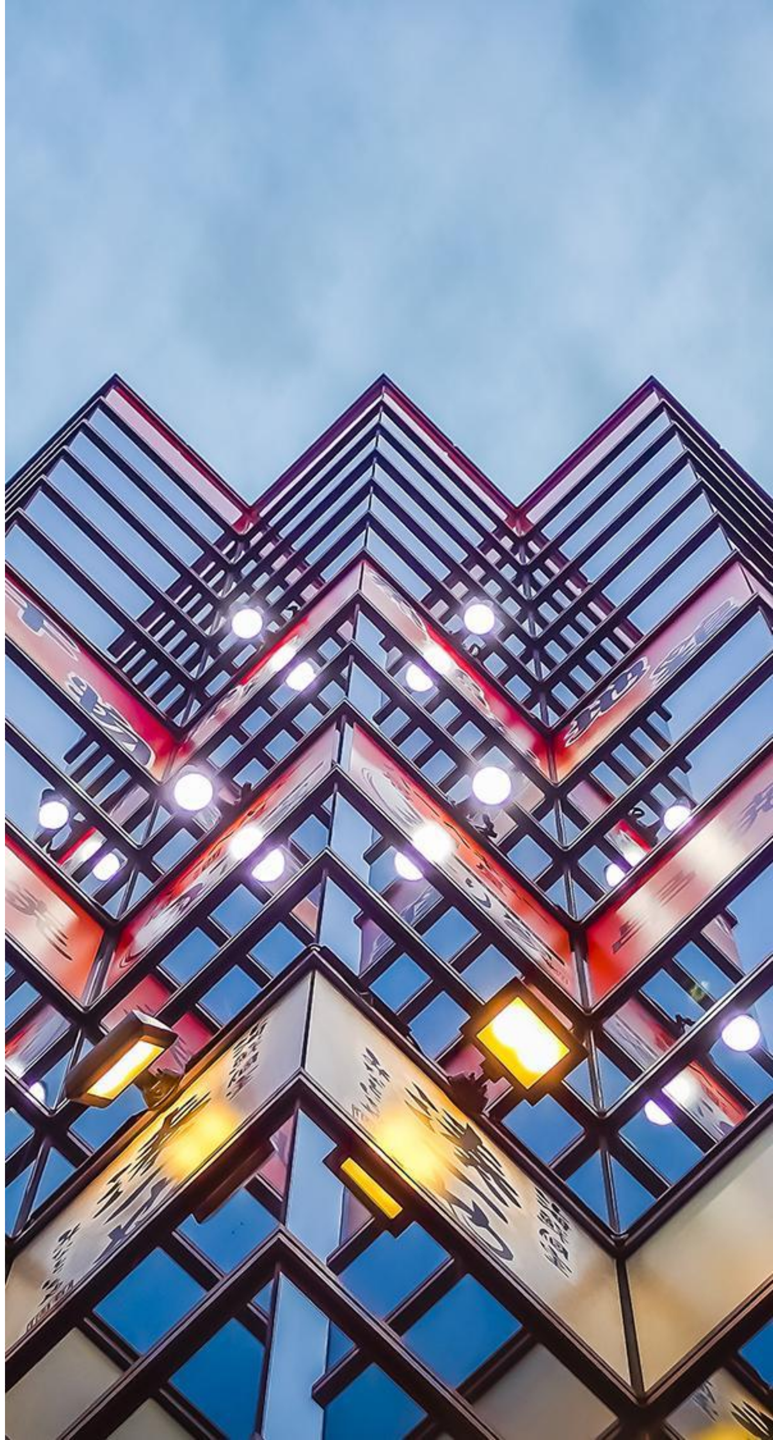
Handle non-movable types

Symantec

```
// wished
DB db("example.db", 100ms, true);
q.emplace(std::nontype<&DB::connect>, std::move(db));

// solved
q.emplace(std::nontype<&DB::connect>,
          std::in_place_type<DB>, "example.db", 100ms, true);
```

What’s my mental model for this feature?



Symantec™

Example from last meeting

```
struct IDoWorkCallback
{
    virtual void OnEvent(WorkResult status, IData &object) = 0;
};

using IDoWorkCallbackPtr = std::shared_ptr<IDoWorkCallback>;

struct WorkContext
{
    void Add(IDoWorkCallbackPtr callback);
};
```

Example: proposed

```
struct WorkContext
{
    typedef void OnEvent(WorkResult status, IData &object);
    void Add(std::function<OnEvent> callback);

    void Add(IDoWorkCallbackPtr callback)
    {
        Add({std::nontype<&IDoWorkCallback::OnEvent>, callback});
    }
};
```

Proposed outcome

```
struct CMyReportingCallback : IDoWorkCallback
{
    void OnEvent(WorkResult status, IData &object) override;
};       Notify

CMyReportingCallback cb;
ctx.Add({std::nontype<&CMyReportingCallback::OnEvent>, cb});
                                            Notify
```

My mental model

```
struct CMyReportingCallback
{
    void Notify(WorkResult status, IData &object);
};

template<invocable<WorkResult, IData &> T>
void Accept(T f);

Accept(cb);
```

A form of concept adaptation

```
struct CMyReportingCallback
{
    void Notify(WorkResult status, IData &object);
};

concept_map invocable<CMyReportingCallback, WorkResult, IData &>
{
    using operator() = CMyReportingCallback::Notify;
};
```

Concept_map → impl block

```
trait Callable<Args> {
    fn call(&self, args: Args);
}

impl Callable<(WorkResult, IData)> for CMyReportingCallback {
    fn call(&self, (status, object): (WorkResult, IData)) {
        self.Notify(status, object);
    }
}
```

A one-time impl block

```
CMyReportingCallback cb;
ctx.Add({std::nontype<
            [](auto &cb, WorkResult status, IData &object)
            {
                LOG(INFO) << "status: " << status;
                cb.Notify(status, object);
            }>,
        cb});
```

An impl block

Symantec

```
template<class T>
inline constexpr auto impl_invocable_for = std::nontype<void>;
template<>
inline constexpr auto impl_invocable_for<CMyReportingCallback> = std::nontype<
    [](auto &cb, WorkResult status, IData &object)
    {
        LOG(INFO) << "status: " << status;
        cb.Notify(status, object);
    }>;
…

ctx.Add({impl_invocable_for<CMyReportingCallback>, cb});
```

Nontype, or constexpr parameter

- nontype<f> is a single-entry witness table passed to a type-erasure at compile-time
- C++ makes switching between passing at runtime vs. at compile-time visible in the language via non-type template parameters
- However, you cannot pass template parameters solely to constructors
- The suggestion is as same as saying making the following equivalent

```
pack.start(std::bind_front(&Conn::send, std::move(obj)));
// and
pack.start(std::bind_front<&Conn::send>(std::move(obj)));
```

Thank you