Life without operator()

Zhihao Yuan

19-Jul-22

I often seen code like this

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

using IDoWorkCallbackPtr = std::shared_ptr<IDoWorkCallback>;

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

Shouldn’t that be…?

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

How to migrate?

```
struct CMyWorkCallback : IDoWorkCallback
{
    void OnEvent(WorkResult status, IData &object) override
    {
        /* ... */
    }
};

ctx.Add(std::make_shared<CMyWorkCallback>());
```

We could add

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

    void Add(IDoWorkCallbackPtr callback)
    {
        Add([=](WorkResult status, IData &object)
            { callback->OnEvent(status, object); });
    }
};
```

Or

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

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

This proposal ↓ P2511

```
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});
    }
};
```

Zero-cost

Signature, not operator()

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

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

The two demands can use one solution

- Existing code accepts interface-based callbacks
- Needs adaptation to switch to type-erased call wrappers that calls only operator()

- Existing code takes type-erased call wrappers
- Needs adaptation to invoke a member function other than operator() with that signature

Let users designate something else as their objects’ operator()

The demands are not unique to C++

- Java

```java
interface DoWorkCallback {
    void OnEvent(WorkResult status, Data object);
}

class WorkContext {
    public void Add(DoWorkCallback callback) {
    }
}
```

Java < 8

```java
interface DoWorkCallback {
    void OnEvent(WorkResult status, Data object);
}

class MyWorkCallback implements DoWorkCallback {
    public void OnEvent(WorkResult status, Data object) {
        /* do work */
    }
}

ctx.Add(new MyWorkCallback());
```

Java 8

SAM (**S**ingle **A**bstract **M**ethod) interface

```
interface DoWorkCallback {
    void OnEvent(WorkResult status, Data object);
}

ctx.Add((status, object) -> {
    /* do work */
});
```

The demands are not unique to C++

- Java
  - There is no operator()
  - Lambda expression creates a class that implements the SAM interface
- C♯

```
delegate void DoWorkCallback(WorkResult status, Data object);
```

C#

```
delegate void DoWorkCallback(WorkResult status, Data object);

class MyWorkCallback {
    public void OnEvent(WorkResult status, Data object) {
        /* do work */
    }
}

var work = new MyWorkCallback();
ctx.Add(work.OnEvent);
```

The demands are not unique to C++

- Java
  - There is no operator()
  - Lambda expression creates a class that implements the SAM interface
- C♯
  - Delegates give signatures names
  - You may pass *wrapped methods*
  - Lambda expression is just another way of specifying a delegate

Going back to C++

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

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

May C++ have wrapped methods?

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

CMyReportingCallback cb;
ctx.Add(cb.Notify);
```

cb.Notify

- Borland C++’s __closure

```
typedef void (__closure *DoWorkCallback)(WorkResult status, IData &object);
```

- Member access as lambda (P0119)

```
[&cb](auto &&...args)
    -> decltype(cb.Notify(std::forward<decltype(args)>(args)...))
{
    return cb.Notify(std::forward<decltype(args)>(args)...);
}
```

The idea is not new

Borland C++’s __closure

- Fat pointer
- Doesn’t accept anything other than member access

Member access as lambda (P0119)

- Closure that captures a reference
- Works poorly when passing to any call wrappers
  - std::function
  - std::move_only_function C++23
  - std::function_ref C++26

Read P2472

The two problems can use one solution

- Force uses of one kind of call wrapper

- Commit to a type that ignores all call wrappers

Solve it inside type-erased call wrappers

Same call-site, different guts

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

- Add(std::function)
- Add(std::move_only_function)

} P2511 (This Paper)

- Add(std::function_ref) → P0792

Beyond operator()

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

Summary

- Let users designate something else as their objects’ operator()
- Solve it inside type-erased call wrappers