On writing loops in PPL and continuation-passing style, part 2

devblogs.microsoft.com/oldnewthing/20230823-00

August 23, 2023

Raymond Chen

Last time, we came up with task-based `while` loop that involved creating a custom callable that passed copies of itself to the next iteration.

This time, we'll implement the function in terms of a more traditional recursion.

```
template<typename Callable>
task<void> do_while_task(
    std::shared_ptr<Callable> const& f)
{
    return (*f)().then([f](bool loop) {
        return loop ? do_while_task(f) :
                      task_from_result();
    });
}

template<typename Callable, typename =
    std::enable_if_t<std::is_invocable_v<Callable>>>
task<void> do_while_task(Callable&& callable)
{
    using Decayed = std::decay_t<Callable>;
    return do_while_task(
        std::make_shared<Decayed>(
            std::forward<Callable>(callable)));
}
```

The real work happens in the first overload, which takes a ready-made `shared_ptr`. The second overload is a convenience method that lets you pass a callable, and it will wrap it in a `shared_ptr` for you.

However, if you think about it, in PPL-style programming, the lambda callable itself usually holds a pointer to shared state, so that the various task fragments can share information with each other. Let's look again at my original example.

```
do_while_task([i = 0, widgets]() mutable
{
    if (i >= 3) return task_from_result(false);
    return create_widget().then([index = i++, widgets](auto widget)
    {
        widgets[index] = widget;
        return true;
    });
}).then([] {
    printf("Done!\n");
});
```

I cheated here and incremented the `i` variable inside the first lambda, but capturing the unincremented value as `index` for the second lambda. More realistically, the two lambdas need to share state.

```
struct lambda_state
{
    lambda_state(Widgets* w) : widgets(w) {}
    Widgets* widgets;
    int i = 0;
};

auto state = std::make_shared<lambda_state>(widgets);

do_while_task([state]()
{
    if (state->i >= 3) return task_from_result(false);
    return create_widget().then([state](auto widget)
    {
        state->widgets[state->i] = widget;
        state->i++;
        return true;
    }
}).then([] {
    printf("Done!\n");
});
```

This means that our `do_while_task` creates a `shared_ptr` that itself holds another `shared_ptr`, which seems kind of silly.

We'll address this next time.