Modern C++ as Concurrent Assembly

Diego Perini

C++Now 2014

Overview

> New Features of C++11/14
> Doppl (a new language)
> Compiled Doppl code (C++ code)

Modern C++

Features

Modern C++

> Better functional programming support
> Concurrency support in standard library
> Better type resolution

> More readable code for the same speed gain

Claim

Concurrency deserves its own type of paradigms and languages

Why so few?

> Concurrency + Safe code is hard

> A Solution:

Functional programming

Immutable data structures

"We all love Haskell but only a few dare to code"

Why so few?

> Portable API for shared memory & message passing is hard to implement

Why so few?

Better Scalability vs Less utility code:

> Scalability always wins!

Claim

Modern C++ has necessary building blocks for a new concurrent language

Ingredients

> Standard library

```
#include <atomic>
#include <functional>
#include <future>
#include <thread>
```

Ingredients

> Parameter packs

```
template<typename C, typename... Ts>
auto call_when_I_need(C callable, Ts... args)
        -> std::future< decltype(callable(args...)) >
{
    return std::async(
        std::launch::deferred, callable, args...
    );
}
```

Ingredients

> Lambdas

  > Suitable for asynchronous operations

  > Can capture environment

```
auto generic_add = [] (auto x, auto y) {
    return x + y;
}
```

Doppl

Data Oriented
Parallel Programming
Language

Doppl

> Proof of concept

    > Compiles to latest C++14 draft (thanks LLVM)

> Natively parallel

> Data oriented

> Readability in mind

Doppl

Natively parallel:

> Built-in parallel types

> Built-in async support

> Shared memory in user space

Doppl

Data oriented:

> Minimal cache miss count

    > Similar data are close to each other in memory

    > Order of placement is the same as order of access

    > Structure of arrays instead of array of structures

Doppl

Readability in mind:

```
task(1) HelloWorld {
    init: {
        output = "Hello World!\n"
    }
}
```

Apologies

Following Doppl code samples are necessary for the sake of argument stated in the beginning

Hello World Inspection

```
task(1) HelloWorld {
    init: {
        output = "Hello World!\n"
    }
}
```

Hello World Inspection

Keyword for defining tasks

```
task(1) HelloWorld {
    init: {
        output = "Hello World!\n"
    }
}
```

Hello World Inspection

Range literal

```
task(1) HelloWorld {
    init: {
        output = "Hello World!\n"
    }
}
```

Hello World Inspection

At least 1, at most 10 copies are allowed

```
task(1 10) HelloWorld {
    init: {
        output = "Hello World!\n"
    }
}
```

Hello World Inspection

Number of CPUs available during compilation

```
task(~) HelloWorld {
    init: {
        output = "Hello World!\n"
    }
}
```

Hello World Inspection

Task name

```
task(1) HelloWorld {
    init: {
        output = "Hello World!\n"
    }
}
```

Hello World Inspection

Reserved keyword for initial state

```
task(1) HelloWorld {
    init: {
        output = "Hello World!\n"
    }
}
```

Hello World Inspection

```
task(1) HelloWorld {
    init: {
        output = "Hello World!\n"
    }
}
```

Global binding for standard output

States

```
task(2) HelloWorld {   #this is a line comment
    init: {
        output = "Hello "
        ->my_state
    }
    my_state: {
        output = "World!\n"
        -->init
    }
}   #infinite hello world loop!
```

States

```
task(2) HelloWorld {   #this is a line comment
    init: {
        output = "Hello "
        ->my_state
    }
    my_
                   "World!\n"
        -->init
    }
}   #infinite hello world loop!
```

Blocking
transition

States

```
task(2) HelloWorld {   #this is a line comment
    init: {
        output = "Hello "
        ->my_state
    }
    my_
                rld!\n"
        -->init
    }
}   #infinite hello world loop!
```

Non-Blocking transition

Members

```
task(10) Members {
    my_int = int

    init: {
        my_int = 42
        my_int = my_int + 1
        finish
    }
}
```

Members

```
task(10) Members {
    my_int = int

    init: {
        my_int = 42
        my_int = my_int + 1
        finish
    }
}
```

Who owns this?

Members

```
task(10) Members {
    my_int = int

    init: {
        my_int = 42
        my_int = my_int + 1
        finish
    }
}
```

Who owns this?

Is this synchronous?

Members

```
task(10) Members {
    my_int = int
    
    init: {
        my_int = 42
        my_int = my_int + 1
        finish
    }
}
```

Who owns this?

Is this synchronous?

Is this synchronous?

Members

```
task(10) Members {
    my_int = int

    init: {
        my_int = 42
        my_int = my_int + 1
        finish
    }
}
```

Who owns this?

Is this synchronous?

Is this synchronous?

What if I want some code to be executed every time I accessed it?

Members

A member declaration clause:

```
scope      monadic    action     member  =  type
semantic   semantic   semantic   name
```

Members

A member declaration clause:

```
scope          monadic        action         member   =  type
semantic       semantic       semantic       name
```

Example:

```
private        just           data           my_int   =  int
```

Members

```
scope      monadic    action     member  =  type
semantic   semantic   semantic   name
```

Available semantics:

```
private    just       data       member  =  type
shared     maybe      future     name
           once       state
           sole       memory
                      element
```

Members

```
scope      monadic    action     member  =  type
semantic   semantic   semantic   name
```

Available semantics:

```
private    just       data       member  =  type
sh                               name



                      element
```

If omitted, defaults are the first line (private, just, data)

Members

```
scope     monadic   action    member  =  type
semantic  semantic  semantic  name
```

Available semantics:

```
private   just      data      member  =  type
shared    maybe     future    name
          once      state
          sole
```

We are interested in these

Scope Semantics

```
task(10) ScopeSemantics {
    private my_int = int              #there are 10 of these
    shared  my_shared_int = int  #there is only 1 of this
    init: {
        my_int = 42
        my_shared_int = 99
        finish
    }
}
```

Monadic Semantics

```
task(10) MonadicSemantics {
    just   my_int = int             #always a value, never null
    maybe  maybe_int = int          #can be null sometimes
    init: {
        my_int = 42
        maybe_int = null
        maybe_int = my_int
        finish
    }
}
```

Action Semantics

```
task(1) ActionSemantics {
   data    my_data   = int
   future  my_future = int
   state   my_state  = int

...
```

Action Semantics

```
init: {
    my_data   = plus_forty_two(3)
    my_future = plus_forty_two(5)
    my_state  = plus_forty_two(my_data)
    finish
}
plus_forty_two(x = int): {
    #some computation that takes time
    yield 42 + x
}
```

Action Semantics

```
init: {
    my_data   = plus_forty_two(3)
    my_future = plus_forty_two(5)
    my_state  = plus_forty_two(my_data)
    finish
}
plus_forty_two(x = int): {
    #some computation that takes time
    yield 42 + x
}
```

Synchronous

Asynchronous

Creates a closure that can be used later.
my_data is captured by value

Action Semantics

```
init: {
    my_data   = plus_forty_two(3)
    my_future = plus_forty_two(5)
    my_state  = plus_forty_two(my_future)
    finish
}
plus_forty_two(x = int): {
    #some computation that takes time
    yield 42 + x
}
```

Is this synchronous?

Action Semantics

```
init: {
    my_data   = plus_forty_two(3)
    my_future = plus_forty_two(5)
    my_state  = plus_forty_two(my_future)
    finish
}
plus_forty_two(x = int): {
    #some computation that takes time
    yield 42 + x
}
```

NO! It's async

`plus_forty_two` adapts its semantics during capture

Compiling Doppl

C++ Code Generation

Compiler Features

> Task range independent runtime

> Scope semantics:

    > private, shared

> Action semantics:

    > data, future, state

> Concurrent stdin and stdout

Assignment and Member Access

- We need a unified interface for any kind of assignment and member access
  - Reason: Avoiding inheritance and pointer type casting

Assignment and Member Access

Assignment:

```
template<typename T, typename... Ts>
void set(T& input, Ts... args);
```

Member Access:

```
template<typename T>
const T& get();
```

Assignment and Member Access

Assignment:

```
template<typename T, typename... Ts>
void set(T& input, Ts&... args);
```

Member Access:

```
template<typename T>
const T& get();
```

We are not putting these in a base class. They are merely concepts. Let's hope concepts become a thing in C++17 or sooner.

Standard Input and Output

C++ Code Generation

Standard Input

```
class input_t {
    private:
    std::mutex m;

    public:
    const std::string get() {
        std::lock_guard<std::mutex> lock(m);
        std::string value;
        std::cin >> value;
        return value;
    };
};
```

Standard Output

```
class output_t {
    private:
    std::mutex m;

    public:
    void set(const std::string& value) {
        std::lock_guard<std::mutex> lock(m);
        std::cout << value;
    };
};
```

Scope Semantics

C++ Code Generation

Private Members

> No encapsulation needed

> Any encapsulation would probably be eliminated by the C++ compiler anyway

Shared Members

> Shared members are bundled with their own mutex

> Shared members forward their set() and get() once the mutex is passed

Attention: Literals must not become Lvalue references during encapsulation

Shared Members

```
template<typename T>
class shared {
private:
      T member;
      std::mutex m;
```

Shared Members

```
public:
    //Forward get
    auto get() -> decltype(member.get()) {
        std::lock_guard<std::mutex> lock(m);
        return member.get();
    };
```

Shared Members

```
public:
    //Assignment for member types
    template<typename V, typename... Vs>
    shared<T>& set(V& v, Vs&... args) {
        std::lock_guard<std::mutex> lock(m);
        member.set(v, args...);
        return *this; //This is for chaining assignments
    };
```

Shared Members

```
public:
    //Assignment for literals
    shared<T>& set(decltype(member.get())& v) {
        std::lock_guard<std::mutex> lock(m);
        member.set(v);
        return *this;
    };
};
```

Action Semantics

C++ Code Generation

Action Semantics

```
enum class semantic_action_specifier {
    data,
    future,
    state/*,
    memory,
    element*/
};
```

Forward Declarations

```
template<semantic_action_specifier S, typename T, typename... Ts>
class task_member;

template<typename T>
using DM = task_member<semantic_action_specifier::data, T>;
template<typename T>
using FM = task_member<semantic_action_specifier::future, T>;
template<typename T, typename... Ts> //Ts is for parameterized states
using SM = task_member<semantic_action_specifier::state, T, Ts...>;
```

Data Members

```
template<typename T>
using DM = task_member<semantic_action_specifier::data, T>;
```

> A template that can store any type

> `set(DM<T>& input)` is a copy assignment

> `set(FM<T>& input)` is a copy assignment, synchronizes input

> `set(SM<T>& input)` is a synchronous call

> `set(T&& input)` is a literal assignment

> `get()` is a synchronous value access

Data Members

```
template<typename T>
class task_member<semantic_action_specifier::data, T> {
private:
    T _data;
public:
    //Get value
    const T& get() {
      return _data;
    };
```

Data Members

```
//data = value
DM<T>& set(T&& input) {
  this->_data = input;
  return *this;
};
```

Data Members

```
//data = data
DM<T>& set(DM<T>& input) {
  this->_data = input._data;
  return *this;
};
```

Data Members

```
//data = state
template<typename... Ts>
DM<T>& set(SM<T, Ts...>& input, Ts&... args)
{
    FM<T> _future(input, true, args...);
    this->set(_future);
    return *this;
};
```

set() method of FM<T> has an is_lazy flag for lazy assignment

Data Members

```
//data = future
DM<T>& set(FM<T>& input) {
  this->_data = input.get();
  return *this;
};
```

Future Members

```
template<typename T>
using FM = task_member<semantic_action_specifier::future, T>;
```

> Future members are interchangeable between threads
>
> > Encapsulates `std::shared_future<T>`

> Value of internal T can be changed by current thread (Sync)

AND

> Value of internal T can be changed by some other thread (Async)
>
> > Encapsulates `std::promise<T>` to retrieve its `std::shared_future`

Future Members

```
template<typename T>
using FM = task_member<semantic_action_specifier::future, T>;
```

- `set(DM<T>& input)` is forwarded to internal promise
- `set(FM<T>& input)` is forwarded to internal shared future
- `set(SM<T>& input, const bool is_lazy)` starts a call
- `set(T&& input)` is a literal assignment and forwarded to promise
- `get()` is a synchronous value access

Future Members

```
template<typename T>
class task_member<semantic_action_specifier::future, T> {
private:
	//Internal future and promise
	std::shared_future<T> _future;
	std::promise<T> _promise;

public:
	//Get value and sync
	const T& get() {
		return _future.get();
	};
```

Future Members

```
//future = data
FM<T>& set(DM<T>& input) {
    std::promise<T> p;
    _future = p.get_future();
    p.set_value(input.get());
    return *this;
};
```

Future Members

```
//future = value
FM<T>& set(T&& input) {
    std::promise<T> p;
    _future = p.get_future();
    p.set_value(input);
    return *this;
};
```

Future Members

```
//future = future
FM<T>& set(FM<T>& input) {
  this->_future = input._future;
  return *this;
};
```

Future Members

```
//future = state
template<typename... Ts>
FM<T>& set(SM<T, Ts...>& input, const bool is_lazy, Ts&... args) {
    _promise = std::promise<T>();
    _future = _promise.get_future();
    auto f = std::async(
        is_lazy ? std::launch::deferred : std::launch::async,
        [&] () { task_loop<T, Ts...>()(input, _promise, args...); }
    );
    if(is_lazy) f.get();
    return *this;
};
```

Future Members

```
//future = state
template<typename... Ts>
FM<T>& set(SM<T, Ts...>& input, const bool is_lazy, Ts&... args) {
    _promise = std::promise<T>();
    _future = _promise.get_future();
    auto f = std::async(
        is_lazy ? std::launch::deferred : std::launch::async,
        [&] () { task_loop<T, Ts...>()(input, _promise, args...); }
    );
    if(is_lazy) f.get();
    return *this;
};
```

This is our runtime loop

Future Members

```
//future = state
template<typename... Ts>
FM<T>& set(SM<T, Ts...>& input, const bool is_lazy, T
    _promise = std::promise<T>();
    _future = _promise.get_future();
    auto f = std::async(
        is_lazy ? std::launch::deferred : std::launch::async,
        [&] () { task_loop<T, Ts...>()(input, _promise, args...); }
    );
    if(is_lazy) f.get();
    return *this;
};
```

This is for yield

This is our runtime loop

State Members

```
template<typename T, typename... Ts>
using SM = task_member<semantic_action_specifier::state, T, Ts...>;
```

> Stores a callable

> Stored callable must be able to transform into a value (yield)

> Stored callable must be able to set the next callable (transition)

>> Encapsulates std::function

> Stored callable must be able to capture some environment

>> Two template instances (captured and uncaptured)

`SM<T>` `SM<T, Ts...>`

State Members

```
template<typename T>
class task_member<semantic_action_specifier::state, T> {
private:
  std::function<
       void(std::promise<T>&, SM<T>&, SM<T>&)
       > _state;
public:
  auto get() -> decltype(_state) {
      return _state;
  };
```

State Members

```
template<typename T>
class task_member<semantic_action_specifier::state, T> {
private:
  std::function<
        void(std::promise<T>&, SM<T>&, SM<T>&)
        > _state;
public:
  auto get() -> decltype(_state) {
      return _state;
  };
```

This is yield

State Members

```
template<typename T>
class task_member<semantic_action_specifier::state, T> {
private:
  std::function<
        void(std::promise<T>&, SM<T>&, SM<T>&)
        > _state;
public:
  auto get() -> decltype(_state) {
      return _state;
  };
```

This is yield

This is next state that we set to transition

State Members

```
template<typename T>
class task_member<semantic_action_specifier::state,
private:
  std::function<
      void(std::promise<T>&, SM<T>&, SM<T>&)
      > _state;
public:
  auto get() -> decltype(_state) {
      return _state;
  };
```

This is yield

This is finish state to end the call

This is next state that we set to transition

State Members

```
public:
SM<T>& set(SM<T>& input) { //state = state  (input is a closure)
    this->_state = input._state;
    return *this;
};

SM<T>& set(decltype(_state) input) { //state = function literal
    _state = input;
    return *this;
};
```

State Members

```
//state = state (input is bound to args to create a closure)
template<typename... Ts, typename... As>
SM<T>& set(SM<T, Ts...>& input, As&... args) {
    this->_state =
        [&] (std::promise<T>& p, SM<T>& n, SM<T>& f) {
            input.get()(p, n, f, args...);
        };
    return *this;
};
```

State Members

```
template<typename T, typename... Ts>
class task_member<semantic_action_specifier::state, T, Ts...> {
private:
    std::function<void(std::promise<T>&, SM<T>&, SM<T>&, Ts...)> _state;

public:
    //Get callable
    auto get() -> decltype(_state) {
        return _state;
    };
```

State Members

```
    //state = state
    SM<T, Ts...>& set(SM<T, Ts...>& input) {
        this->_state = input._state;
        return *this;
    };

    //state = function literal
    SM<T, Ts...>& set(decltype(_state) input) {
        _state = input;
        return *this;
    };

};
```

Runtime (aka Task Loop)

C++ Code Generation

Task Loop

> A loop that handles state transition

> A loop that never throws exceptions

> Its initial state should accept parameters

Task Loop

> Other than some internal handles, it should not pollute its creators' stack

> Reason: We should be able to create task loops as many as we want without worry during runtime

> Reason: Temporarily cloned tasks (remember `future=state` assignments) must use its creators' members without copy.

Task Loop

```
template<typename T, typename... Ts>
class task_loop {
private:
    SM<T> initial_state;
    SM<T> next_state;
    SM<T> finish_state;

    std::future<void> current_state_future;
```

Yield type

Arguments for initial state

Task Loop

```
public:
int operator() (SM<T, Ts...>& init, std::promise<T>& yield,
                Ts&... args) {
    //Set init
    initial_state.set(init, args...);

    //Infinite loop escape trigger
    bool is_running = true;
    finish_state.set(
    [&is_running] (std::promise<T>& yield, SM<T>& next, SM<T>& finish)
        { is_running = false; } );
```

Task Loop

```
//Default next state assignment
next_state.set(finish_state);

//Initial state assignment (lazy)
current_state_future = std::async(
    std::launch::deferred,
    [&] () {
        initial_state.get()(yield, next_state, finish_state);
    }
);
```

Task Loop

```
//Program loop
try {
    while(is_running) {
        //Run lazily assigned state code
        current_state_future.get();
        //Set next state (lazy)
        current_state_future = std::async(
            std::launch::deferred, [&] () {
            next_state.get()(yield, next_state, finish_state);
        });
    }
}
```

Task Loop

```
    //Error
    catch(std::exception& e) {
        return 1; //Or abort, dunno :)
    }

    return 0;
};
```

Future Members (previous slide)

```
//future = state
template<typename... Ts>
FM<T>& set(SM<T, Ts...>& input, const bool is_lazy, T
    _promise = std::promise<T>();
    _future = _promise.get_future();
    auto f = std::async(
        is_lazy ? std::launch::deferred : std::launch::async,
        [&] () { task_loop<T, Ts...>()(input, _promise, args...); }
    );
    if(is_lazy) f.get();
    return *this;
};
```

This is for yield

This is our runtime loop

int main()

C++ Code Generation

Pseudo Code

> Declare *range* as const int

> Declare stdin and stdout as input_t and output_t

> Declare shared members as shared

> Declare std::array<T, range> for each private member

> Declare *task_body* as lambda

    > Declare a task_loop

    > Declare states

        > Declare local members

        > Insert program code

    > Start declared loop

> Call std::async(*task_body*) *range* times

Future Studies

Wrapping Other Modern C++ Components

Future Studies

> Partially applied closures: std::bind

> Native tuple type: std::tuple (see project site)

> Collections: std::array, std::vector ...

> Element members: Iterators

> Memory members: std::shared_ptr, std::unique_ptr

> Once members: std::call_once (see project site)

> Sole members: std::lock_guard

Doppl

> Website:

  doppl.org

> Presentation Materials and Doppl runtime:

  github.com/diegoperini

> Contact:

  diego@dperini.com

Questions?

Thank you!