Functional Programming

A case study of C++ and JavaScript

LI, Kwan To

COMP 4111 Independent Study

April 22, 2020

Agenda I

Agenda II

4 Unique features of C++/JavaScript FP
- Functional Objects
- Non-pure Functions and Closures

5 FP Frameworks and Libraries
- Lodash Utility Library
- Immer
- Redux

6 Conclusion

Introduction

Introduction

What is Functional Programming (FP)?

What is Functional Programming?

What is functional programming (FP)?

- no strict definition
- a programming paradigm
- a style of programming using functions mainly
  - instead of expressions or declarations
- running a program - evaluation of the function

Comparison

A high-level comparison:

	Object-oriented Programming	Functional Programming
model	real-life objects	mathematical functions
program	interactions among objects	compositions of functions
memory	object fields	stateless

Introduction

Lambda Calculus

Origin

Functional programming is originated from lambda calculus.
You may have seen lambdas before...

Lambda Expressions (Lambdas)

Syntax: Lambda

[<capture-list>] (<parameter-list>) mutable →<return-type> { <body> }

- C++11's lambda expressions enable you to define anonymous functions — functions *without* a name.
- They are defined locally inside functions.
- The capture list (of variables) allows the lambda to use the local variables that are already defined in the enclosing function.
  - [=]: capture all local variables by value.
  - [&]: capture all local variables by reference.
  - [variables]: specify only the variables to capture
- The return type
  - is void by default if there is no return statement.
  - is automatically inferred if there is a return statement.
  - may be explicitly specified by the → syntax.

Example: Simple Lambdas with No Captures

```
#include <iostream>     /* File : simple-lambdas.cpp */
using namespace std;

int main()
{
    // A lambda for computing squares
    int range[] =  { 2, 5, 7, 10 };
    for (int v : range)
        cout << [](int k) { return k * k; } (v) << endl;

    // A lambda for doubling numbers
    for (int& v : range) [](int& k) { return k *= 2; } (v);
    for (int v : range) cout << v << "\t";
    cout << endl;

    // A lambda for computing max between 2 numbers
    int x[3][2] = { {3, 6}, {9, 5}, {7, 1} };
    for (int k = 0; k < sizeof(x)/sizeof(x[0]); ++k)
        cout << [](int a, int b) { return (a > b) ? a : b; } (x[k][0], x[k][1])
             << endl;

    return 0;
}
```

Lambda Calculus

Mathematically, lambda calculus has only 3 types of building blocks:

1. variables: a parameter or logical value (x)
2. abstraction: the definition of a function ($\lambda x.M$)
3. application: applying a function - substituting a term into an abstraction (MN)

Example

$\lambda x.x$ is the identity function

Example

$(\lambda x.x\ x)(\lambda x.x\ x)$ is a program that loops forever.

The lambda expressions in programming languages are basically abstractions in lambda calculus.

Building Blocks of a Functional Language

We can build up programs (functions) with lambda calculus.

Boolean Logic

We can define TRUE, FALSE and logical operator like the following:

- $TRUE = \lambda x.\lambda y.x$
- $FALSE = \lambda x.\lambda y.y$
- $AND = \lambda p.\lambda q.p\ q\ p$
- $OR = \lambda p.\lambda q.p\ p\ q$
- $IFTHENELSE = \lambda p.\lambda a.\lambda b.p\ a\ b$

Recursion

The Y combinator allows us to apply a function over and over again.

$$Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

Building Blocks of a Functional Language

Arithmetic

We can define numbers as the number of times that we apply a certain function:

- $0 = \lambda f.\lambda x.x$
- $1 = \lambda f.\lambda x.f\ x$
- $2 = \lambda f.\lambda x.f\ (f\ x)$
- $SUCC = \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$
- $ADD = \lambda m.\lambda n.m\ SUCC\ n$

Introduction

Functional Programming Languages

Functional Programming Languages

There are many programming languages implementing FP:

- Functional languages
  - Haskell
  - SML
- Imperative languages that support FP
  - C++
  - Java
  - JavaScript
  - Python

Features of Functional Programming

Features of Functional Programming

Recursion

Recursion

Definition

Recursion, or self-application, is a function that calls itself.

Examples

The Fibonacci number is defined as:

$$F(n) = \begin{cases} F(n-1) + F(n-2) & \text{if } n \geq 2 \\ n & \text{otherwise} \end{cases}$$

Most programming languages (not only the functional ones) implements recursion.

Examples

A simple bash script:

```
:(){ :|:& };:
```

How would a functional language different from an imperative language then? There does not exist concepts like loops in functional languages!

Imparative

```
function mySum(n) {
    var curSum = 0;
    for (var i = 1; i <= n; i++)
        curSum += i;
    return curSum;
}
```

Functional

```
function mySum(n) {
    if (n == 0)
        return 0;
    else
        return mySum(n - 1) + n;
}
```

Sidenote: In fact, the code for functional in this slide is written in a ’not too functional’ style (will see why later).

Recursion

Coding in a recursive style can remove the need of states.

That is, we do not need to store the intermediate computational results explicitly.

From the above example, the result of the function `mySum(n)` only depends on the input argument `n`.

We call this kind of functions pure functions.

Features of Functional Programming

Pure functions

Pure functions

Definition

A function is considered pure if it has the following properties:

1. any calls with same input arguments produce the same return value
2. has no side-effects

In other words, the result of the computation of a pure function does not depend on the memory/IO and the execution of it would not affect the memory/IO.

Example

A mathematical function is pure.

```
int f(int x, int y) {
    return x + 2 * y;
}
```

Examples

Example

The function is not pure because it refers from the global variables.

```
int z;
int f() {
    return z;
}
```

Example

The function is not pure because it modifies global variables.

```
int z;
int f(int x) {
    z = x;
    return 0;
}
```

Examples

Example

The function is not pure because it involves I/O.

```
int f() {
    int z; cin >> z;
    return z;
}
```

Example

The function is not pure because it mutates and depends on static variables.

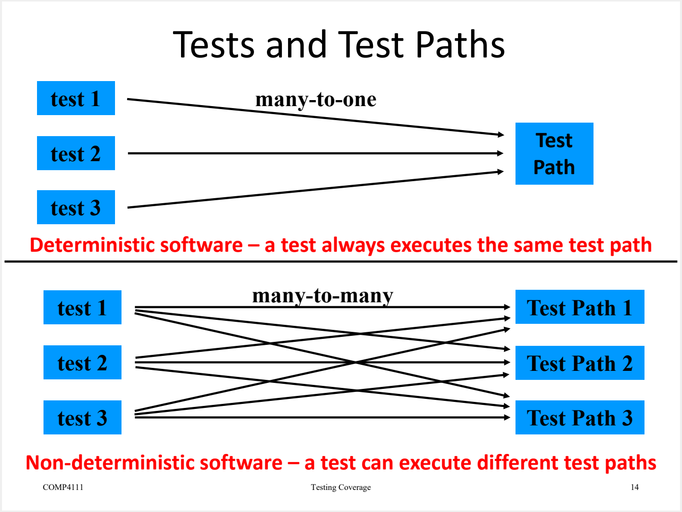
```
int f() {
    static int z = 0;
    return ++z;
}
```

Advantages

Pure functions are the essence of functional programming. This unique feature of functional programming brings numerous benefits:

1. Parallel computing (more on this later)
2. Improved readability
3. Easier to debug

A pure function returns a consistent result for some given arguments. The behaviour of such functions is deterministic.



Tests and Test Paths

Deterministic software – a test always executes the same test path

Non-deterministic software – a test can execute different test paths

COMP4111 Testing Coverage 14

Features of Functional Programming

Higher-order functions

Higher-order functions

Definition

A higher-order function is a function that

1. takes a function as an argument; and/or
2. returns a function as its result

Without we knowing it, we have seen examples of higher other functions before (in mathematics).

Example

The differential operator is a higher-order function:

$$\frac{d}{dx}x^2 = 2x$$

Function composition

To see why higher-order function is interesting in functional programming, we need to see some more examples.

Example

The function composition operator is a higher-order function:

$$(f \circ g)(x) = f(g(x))$$

Function composition is quite common in programming, especially in a pipeline. Let's say you are writing a shell script, you may want to write an alias for `cat $x | grep warning | head`.

Map

In vector/matrix algebra, we often consider transformations. We often apply a function (usually multiplying with an matrix) to transform one point to another.

Map

Consider a weird transformation function $g : \mathbb{R}^n \to \mathbb{R}^n$.

$$g(\begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}^T) = \begin{bmatrix} f(x_1) & f(x_2) & \dots & f(x_n) \end{bmatrix}^T$$

where $f : \mathbb{R} \to \mathbb{R}$ is a arbitrary function defined on $\mathbb{R}$.

So, how do we describe the function the produce g given some specific f.

Example

The function m that vectorize the function f is a higher-order function.

$$m : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R}^n \to \mathbb{R}^n)$$

`map` is a common higher-order function in functional programming. Unlike the above transformation, it is usually more general and can be applied to most vector-like types (e.g. collections, array, tuples).

Map

Unlike applying the function one by one using a for loop in imperative programming, `map` enables the possibility to run the function in parallel. However, certain requirements must be met:

- The functions can be executed out of order
- The functions have no inference on another

A pure function in functional programming satisfies the above requirements.

Curry

Definition

Currying means constructing a new function $h : X \to (Y \to Z)$ from an arbitrary $f : (X \times Y) \to Z$ where

$$f(x, y) = h(x)(y)$$

That is,

$$curry(f) = h$$

Currying allows us to do partial application. Let's see an example next slide.

Curry

Example

Assume we have $add(x, y) = x + y$. We can define a curry version of it as $cadd = curry(add)$.

$$\begin{aligned} cadd(x)(y) &= x + y \\ cadd(3)(y) &= 3 + y \\ cadd(3)(4) &= 3 + 4 = 7 \end{aligned}$$

If we additional define $add3 = cadd(3)$, we have

$$\begin{aligned} add3(y) &= 3 + y \\ add3(4) &= 3 + 4 = 7 \end{aligned}$$

Other Higher-order Functions

- `filter` - filter out elements according to certain predicate
- `reduce/fold` - combine the elements using a certain function (e.g. add)
- `apply` - applying a certain function with specifc arguments

C++/JavaScript Implementation of FP

C++/JavaScript Implementation of FP

Lambdas

LAMBDAS IN C++

In C++, lambdas are anonymous functions. It includes:

1. capture list (more on this later)
2. parameter list
3. function body

Example

All three statements will output 3.

```
cout << [](int x, int y) { return x + y; } (1, 2) << endl;
cout << [](auto p) { return p(1); } ([](int x) {return x + 2; }) << endl;
cout << [](int x) { return [x](int y) { return x + y; }; } (1)(2) << endl;
```

Lambdas in C++

Each lambda has its own unique class type called ClosureType. The class has the following defined:

- operator() - runs the function body (and returns the result)
- user-defined operator - allows implicit conversion to function pointers [1]
- defaulted copy constructor
- deleted default constructor and copy assignment operator
- captured variables - as class fields

Example

This program will output 5.

```
template <class T> void f(T g) { cout << g(5) << endl; }
int main() { f([](int x) { return x; }); return 0; }
```

[1]only defined when there is no capture

Lambdas in JavaScript

JavaScript lambdas are simpler. They do not have capture list but capture all variable in the same scope (function).

Example

All of the three statements return 3.

```
((x, y) => x + y)(1, 2);
(p => p(1))(x => x + 2);
(x => y => x + y)(1)(2);
```

Due to the fact that JavaScript is not a strongly-typed language, its syntax is much cleaner.

C++/JavaScript Implementation of FP

Higher-order Functions

C++ Standard Library

The C++ standard library provides quite a number of higher-order functions. However, most of them are not written in a functional style - directly mutating the container.

Map in C++

An implementation of map in C++ would be `std::transform`.

std::transform

```
template< class InputIt, class OutputIt, class UnaryOperation >
OutputIt transform( InputIt first1, InputIt last1, OutputIt d_first,
                    UnaryOperation unary_op );
```

- `first1`, `last1` - range to apply the transform
- `d_first` - the iterator pointing to the beginning of the output
- `unary_op` - an unary function (can be lambda/ functional object)

Example

This program will output `2 3 4`.

```
std::vector<int> v{1, 2, 3}, u(v.size());
std::transform(v.begin(), v.end(), u.begin(), [] (int x) { return x + 1; });
for (int x : u) std::cout << u << ' ';
```

Reduce in C++

An implementation of reduce in C++ would be `std::reduce`.

std::reduce

```
template<class InputIt, class T, class BinaryOp>
constexpr T reduce(InputIt first, InputIt last, T init, BinaryOp binary_op);
```

- `first`, `last` - range to apply the transform
- `binary_op` - a binary function that "sums" the range

Example

This program will output 3.

```
std::vector<int> v{2, 3, 5};
std::cout << std::reduce(v.begin(), v.end(), 0,
                         [] (int x, int y) { return (x + y) % 7; })
          << std::endl;
```

Filter in C++

An implementation of filter in C++ would be `std::copy_if`.

std::copy_if

```
template< class InputIt, class OutputIt, class UnaryPredicate >
OutputIt copy_if( InputIt first, InputIt last,
                  OutputIt d_first, UnaryPredicate pred );
```

- `first`, `last` - range to filter
- `d_first` - the iterator pointing to the beginning of the output
- `pred` - a predicate that only elements evaluated to `true` will be kept

Example

This program will output `2 4 6`.

```
std::vector<int> v{2, 3, 4, 5, 6}, u;
std::copy_if(v.begin(), v.end(), back_inserter(u),
             [] (int x) { return std::gcd(x, 4) > 1; });
for (int x : u) std::cout << x << ' ';
```

Other STL functions

There are more STL functions that support the use of lambdas/functional objects. You can always refer to cppreference for more details.

Example

- `std::sort` - sort according to certain predicate
- `std::find_if` - find certain element fulfilling the predicate
- `std::apply` - partially apply function arguments

Functional style in C++

We can create wrappers to STL functions to write in a more functional way.

Example

Below shows an implementation of map and reduce in curried pure function.

```cpp
template <class Func>
auto myreduce(Func f) {
        return [&f] (int init) {
                return [&f, &init] (const std::vector<int> &v) {
                        return std::reduce(v.begin(), v.end(), init, f);
                };
        };
}

template <class Func>
auto mymap(Func f) {
        return [&f] (const std::vector<int> &v) {
                std::vector<int> u(v);
                std::transform(v.begin(), v.end(), u.begin(), f);
                return u;
        };
}
```

Functional style in C++

Example

The program below will output 6.

```
int main() {
        auto addm7 = [] (int x, int y) { return (x + y) % 7; };
        auto add1 = [] (int x) { return x + 1; };
        std::cout << myreduce(addm7)(0)(mymap(add1)({2, 3, 5}))
                << std::endl;
        return 0;
}
```

The examples above are proof-of-concepts and an efficient/generic implementation are much more complicated.

Parallel Execution

In C++17 or above, execution policies can be specified to speed up computation for some STL functions.

Example

The non-parallel version runs in 340 ms.

```
const int N = 10000000;
std::vector<int> v(N); std::iota(v.begin(), v.end(), 0);
std::cout << std::reduce(v.begin(), v.end(), 0, std::gcd<int, int>)
          << std::endl;
```

The parallel version runs in 200 ms.

```
const int N = 10000000;
std::vector<int> v(N); std::iota(v.begin(), v.end(), 0);
std::cout << std::reduce(std::execution::par,
                         v.begin(), v.end(), 0, std::gcd<int, int>)
          << std::endl;
```

Both program output a single integer 1.

Compiler Support

This feature is quite new. Please check the language support of your compiler. For GCC 9, you need to link the required library with the flag `-ltbb`.

JavaScript Array Class

A set of higher-order functions are defined in the Array class prototype in JavaScript. Unlike C++, most functions return a copy of the array and do not modify the original one.

Map, Reduce and Filter in JavaScript

The standard `map`, `reduce` and `filter` functions are available in JavaScript. All three functions are member functions (prototype) of the `Array` class. They accept a single argument which is the function needed for the operation.

Example

```
[1, 3, 5].map(x => x + 2) // [3, 5, 7]
[2, 3, 5].reduce((x, y) => (x + y) % 7) // 3
[2, 3, 5, 7, 8].filter(x => x % 2 == 0) // [2, 8]
```

As you can expect, the code is much cleaner than C++.

JavaScript FP Example

You can do many cool stuffs with those functions, like writing quicksort in a single line!

Example

The below program runs in 1 second.

```
var mysort = x => x.length > 1 ?
             mysort(x.filter(z => z < x[0]))
               .concat(x.filter(z => z == x[0]))
               .concat(mysort(x.filter(z => z > x[0]))) :
             x;

console.log(mysort([5, 6, 3, 1, 2, 3, 4, 7]));
// [1, 2, 3, 3, 4, 5, 6, 7]

var randArr = Array(1000000).fill().map(x => Math.random());
// 1 million random numbers
console.log(mysort(randArr).slice(0, 5));
```

Other JavaScript functions

Other functional components in Vanilla JS includes:

- `Array.prototype.find()`
- `Array.prototype.every()`
- `Function.prototype.apply()`
- `Function.prototype.bind()`
- `Function.prototype.call()`

There are JS frameworks and libraries that provide more functional components.

Unique features of C++/JavaScript FP

Unique features of C++/JavaScript FP

Functional Objects

Object-oriented Language

Both C++ and JS are object-oriented languages. In both languages, the lambda created are objects.

- In C++, lambdas created have a unique `ClosureType` class
- In JavaScript, lambdas created are `Function` objects

We can "group" the lambda objects in an array or another object to do interesting operations.

C++ Operator Overloading

The ( ) operator in C++ can be overloaded. In fact, for all objects with class that have the respective ( ) operator defined can be used in STL functions.

Example

The output of the following program is 6 5.

```
class Foo {
        private:
                int x;
        public:
                Foo(int x) : x(x) {}
                int operator()(int l, int r) { return (l + r) % x; }
};

int main() {
        std::vector<int> v{2, 5, 6};
        std::cout << std::reduce(v.begin(), v.end(), 0, Foo(7)) << ' ';
        std::cout << std::reduce(v.begin(), v.end(), 0, Foo(8)) << std::endl;
        return 0;
}
```

`ClosureType` has the ( ) operator defined (see previous slides).

STD::FUNCTION CLASS

The `std::function` class is used for generalizing any types with specific argument types and returns types. So, different lambdas (with different `ClosureTypes`) can be group together and carry out operations.

Example

The output of the following program is 3.

```
std::vector<std::function<int(int)>> v;
v.emplace_back([] (int x) { return x + 5; });
v.emplace_back([] (int x) { return x * 3; });
v.emplace_back([] (int x) { return x % 6; });
int res = 0;
for (auto f : v) res = f(res);
std::cout << res << std::endl;
```

JavaScript Function as Objects

In JavaScript, you can use lambdas like any other objects.

Example

```
var calc = {
    add: x => y => x + y,
    minus: x => y => x - y,
    mult: x => y => x * y
};

for (let [k, v] of Object.entries(calc))
    console.log(${k}: ${v(3)(5)});
```

Example

```
var vec = [x => x + 1, x => x * 2, x => x - 6];
console.log(vec.reduce((f, g) => x => g(f(x)))(3)); // 2
```

Unique features of C++/JavaScript FP

Non-pure Functions and Closures

Imperative Language with Functional Features

Both C++ and JavaScript are imperative languages. How do we leverage the advantages of imperative programming when using functional components?

- state of execution
- side-effects

Pass by Reference

In C++, arguments can be passed by reference. This is still the case for lambda expressions. We can use higher-order functions like `std::for_each` to apply a function over certain range.

Example

```
std::vector<int> v{2, 3, 5};
std::for_each(v.begin(), v.end(), [] (int &z) { z--; });
for (int x : v) std::cout << x << ' ';
```

The program will output `1 2 4`.

C++ Lambda Captures

In C++ lambda expression, we can set specify a set of variables in the current scope to be captured inside the lambda.

Example

We can sort job indices according to their finishing time.

```cpp
const int N = 5;
std::vector<int> job_ind(N); std::iota(job_ind.begin(), job_ind.end(), 0);
std::vector<int> job_sta{1, 0, 2, 4, 5};
std::vector<int> job_fin{3, 2, 9, 8, 6};
std::sort(job_ind.begin(), job_ind.end(),
          [&job_fin] (int l, int r) { return job_fin[l] < job_fin[r]; });
for (int i : job_ind) std::cout << i << ' ';
```

Note that the lambda function is not pure as the result differs for different finishing time array.

JavaScript Lambdas

When creating a lambda, JavaScript captures all the variables in the same lexical scope (function scope) by default.

Example

Here, the array `arr` is captured in the lambda.

```
var arr = [1, 2, 4];
[0, 2].map(x => arr[x]); // [1, 4]
```

JavaScript Closures

Recall JavaScript variables uses function scope by default. To access an variable inside a function, we can return a function that access that particular variable. When creating such functions, the lexical environment within which that function was declared is also returned. So, we can say a closure (the combination of the two) is returned.

Example

```
function rand() {
        var x = Math.random();
        return l => r => l + (r - l) * x;
}

var myRand = rand(); // construct a new random
console.log(myRand(0)(2)); // 0.9198049761488316
console.log(myRand(50)(70)); // 59.19804976148832
var myRand2 = rand(); // construct another random
console.log(myRand2(0)(2)); // 1.0531786562191559
console.log(myRand2(50)(70)); // 60.531786562191556
```

Note that only a new random value (x variable) is generated for each `rand( )` call.

Generator Pattern

A common pattern in programming would be the generator pattern. The closure returned generates a new value on demand.

Example

The generator generates $init * 2^k$ for $k = 1, 2, \ldots$.

```
function gen(init) {
        var n = init;
        return () => n = n + n;
}

var myPow = gen(3);
console.log(myPow()); // 6
console.log(myPow()); // 12
console.log(myPow()); // 24
console.log(myPow()); // 48
```

Sidenote: In newer version of JavaScript, there is build-in support of generator (`function*`).

FP Frameworks and Libraries

FP Frameworks and Libraries

Lodash Utility Library

Limitation of Vanilla JS

Vanilla JS is an imperative language, it is not suitable to write code in a purely functional way.

- only limited to few classes (Array)
- written in a mixed OO-style
- functions not curried by default

Lodash FP module

Lodash is a utility library that provides quite a number of utility functions. Its FP modules provides a version of those utility functions tailored for functional programming.

Example

```
var fp = require('lodash/fp');
fp.map(x => x + 2)([1, 2, 4]); // [3, 5, 6]
fp.reduce((x, y) => (x + y) % 7)(0)([2, 3, 5]) // 3
```

Note that the functions are curried functions which allows partial application.

Example

```
var fp = require('lodash/fp');
var vadd2 = fp.map(x => x + 2);
vadd2([2, 3, 3]); // [4, 5, 5]
vadd2([4, 5]); // [6, 7]
```

Curry in JavaScript

You can create you own curried function with the `curry` function. The curried function is created in a way that it can accept one or more arguments at once.

Example

```
var fp = require('lodash/fp');
var fun = (x, y, z) => x + y - z;
var cfun = fp.curry(fun);
cfun(2)(5)(6); // 1
cfun(2)(5, 6); // 1
cfun(2, 5)(6); // 1
cfun(2, 5, 6); // 1
```

It also allows placeholders for finer control on which arguments to apply.

Example

```
var _ = require('lodash/fp');
var fun = (x, y, z) => x + y - z;
var cfun = _.curry(fun);
cfun(2, _, 5)(6); // 3
cfun(_, _, 6)(3, 1); // -2
```

Function Compositions

In functional programming, we define a sequence of actions to apply instead of applying a chain of operations. The library provides `compose` for composing functions.

Example

```
var fp = require('lodash/fp');
var action = fp.compose([
        fp.reduce((x, y) => (x + y) % 7)(0),
        fp.filter(x => x > 4),
        fp.map(x => x * x)
]);
console.log(action([2, 3, 5])); // 6
console.log(action([1, 2, 7])); // 0
```

Consider Vanilla JS, you need to create a function that wraps the whole chain of actions if you want to apply the action repeatedly. Does function composing sound more logical and convenient?

FP Frameworks and Libraries

Immer

Immutable Data Structures

Immutable data structures, with the support of persistence, has become more popular in the recent decade.

- Efficiency - allocate new instance quickly by copy on write
- Concurrency - avoid races in multi-threaded process

Immer Examples

For each update operation for immer data structures, a new container is returned.

Example

```
#include <immer/vector.hpp>
int main()
{
    const auto v0 = immer::vector<int>{};
    const auto v1 = v0.push_back(13);
    assert(v0.size() == 0 && v1.size() == 1 && v1[0] == 13);

    const auto v2 = v1.set(0, 42);
    assert(v1[0] == 13 && v2[0] == 42);
}
```

Example is from `https://sinusoid.es/immer/introduction.html`.

Remember the wrapper we wrote earlier for making STL functions functional? We can now write it in a more efficient way with immer, which avoid the cost of allocating new containers.

Example

This functions insert a range of values to the end of the container. Of course, you can write it in a more functional way by currying.

```
immer::vector<int> myiota(immer::vector<int> v, int first, int last)
{
    for (auto i = first; i < last; ++i)
        v = std::move(v).push_back(i);
    return v;
}
```

Example is from `https://sinusoid.es/immer/containers.html`.

We will see cases that immutable data structures are useful in the next framework introduced.

FP Frameworks and Libraries

Redux

State Management Problem

When you application starts to scale up...

- the number of variables increases a lot
- need to consider synchronization issues
- more complex logic and data flow

The application states would be difficult to manage and test, especially with functions with side-effects.

Redux breaks down the management problem to only three pieces:

- state - all information to describe the current execution of the application
- actions - an object that used to change the state
- reducer - function that takes state and action as arguments

Example

Sample codes can be found at `https://redux.js.org/introduction/core-concepts`.

Three Principles

Redux can be described in three fundamental principles:

- Single source of truth - states are stored in a centralized store
- Immutable states - get a new state via dispatching an action to the store
- Pure reducers - reducers are pure functions

The three principles makes debugging very easy. One can easily trace back the states from the store and easily reproduce the behavior given a single state.

Example

You can read the detailed concepts at `https://redux.js.org/introduction/three-principles`.

Async Calls

You may want to handle asynchronous calls with a middleware such as Redux Thunks or Redux Saga.

Conclusion

Conclusion

For me, functional programming is

- a style of programming - no paradigm is the best
- to write efficient and clean pipelined program
- non-mathy mathematics - lambda calculus
- complicated but interesting (think about monads)

What about you?

Reference Sites

Most materials are based on language references and my personal experience.

- cppreference:
  `https://en.cppreference.com/w/`
- MDN:
  `https://developer.mozilla.org/en-US/docs/Web/JavaScript`
- Wikipedia:
  `https://www.wikipedia.org/`
- Lodash:
  `https://lodash.com/`
- Immer:
  `https://github.com/arximboldi/immer`
- Redux:
  `https://redux.js.org/`

Sample Codes

You can find codes used in this presentation in the following repo:
`https://github.com/STommydx/comp4111-presentation`

Thank You

Thank you!

Any questions?