Міністерство освіти і науки, молоді та спорту України
Національний університет “Львівська політехніка”

Кафедра ЕОМ

Звіт

з домашнє завдання №27.3 з дисципліни:
“Алгоритми та моделі обчислень”
Варіант: № 24.

Виконав:
ст. групи КІ-203
Ширий Богдан Ігорович
Перевірив:
ст. викладач кафедри ЕОМ
Козак Назар Богданович

Львів – 2023

ЗАВДАННЯ:

УМОВА:

Виконати домашнє завдання №27.1 повторно за допомогою мови C++ використовуючи RxCpp (реалізація ReactiveX для C++)

ВИБІР ВАРІАНТУ:

$$(N_{ж} + N_{г} + 1)\%30 + 1 = (24 + 3 + 1)\%10 + 1 = 28\%10 + 1 = 9,$$

де: $N_{ж}$ – порядковий номер студента в групі, а $N_{г}$ – номер групи.

Отож, мій шуканий варіант – це 9 можливих спроб введення ключа валідації.

ВИКОНАННЯ:

Склав C++ програму та зобразив її роботу на рисунку 1.

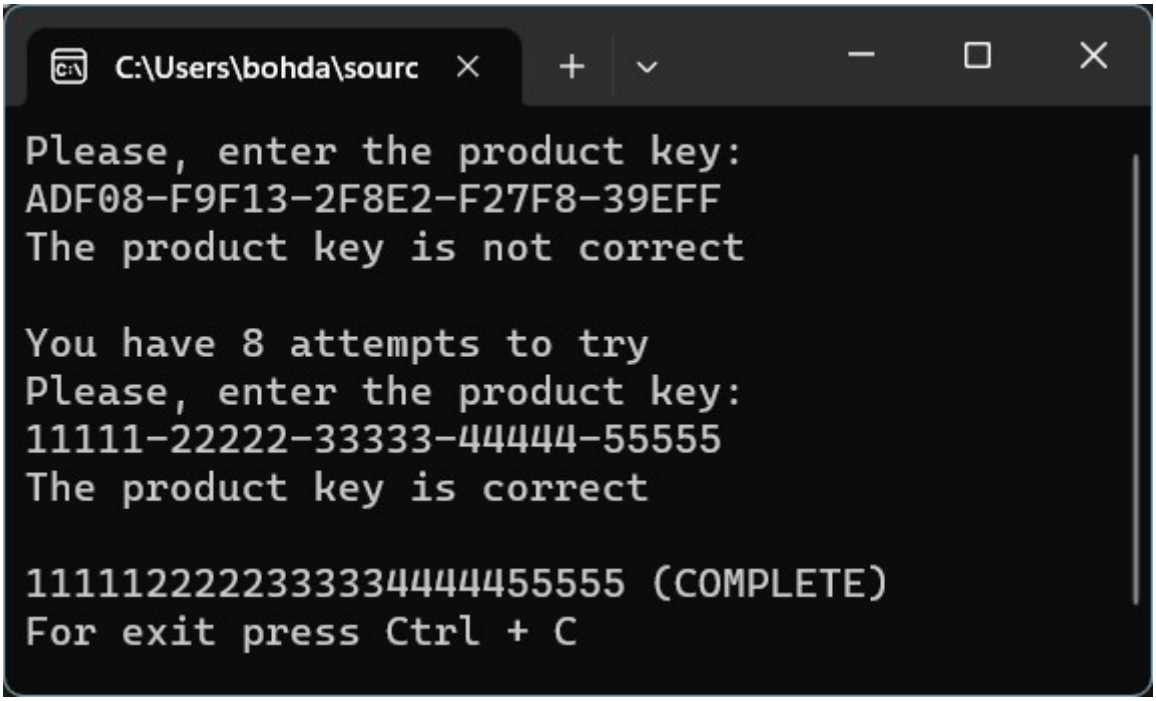

Рис. 1. Виконання програми написаної на C++.

Відповідно, у лістингу 1 навів код програми написаної на C++:

Лістинг 1. Код програми написаної на C++.

```
#define _CRT_SECURE_NO_WARNINGS
#define WIN32_LEAN_AND_MEAN

#ifndef TO_RXCPP_IMPLEMENTATION
#define TO_RXCPP_IMPLEMENTATION
#include "acmShyryiHW27_3.cpp"

#if _WIN32
#include <conio.h>
#include <Windows.h>
#pragma comment(lib, "Ws2_32.lib")
```

```
#pragma comment(lib, "psapi.lib") // #pragma comment(lib, "Kernel32.lib")
#pragma comment(lib, "Iphlpapi.lib")
#pragma comment(lib, "userenv.lib") // Userenv.lib
#endif

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#if __linux__
#include <termios.h>
#include <unistd.h> // //
#endif

//#include "rxcpp/rx.hpp"
namespace rx = rxcpp;
namespace rxsub = rxcpp::subjects;
namespace rxu = rxcpp::util;
#include <cctype>
#include <clocale>

#define ATTEMPTS_COUNT 9
int attemptsDownCount = ATTEMPTS_COUNT;

#define GROUPS_DIGITS_COUNT 5
#define GROUP_DIGITS_SIZE 5

const unsigned char PRODUCT_KEY_PART1[] = {
        0xF, 0xF, 0xF, 0xF, 0xF,
        0xD, 0xD, 0xD, 0xD, 0xD,
        0x8, 0x8, 0x8, 0x8, 0x8,
        0xB, 0xB, 0xB, 0xB, 0xB,
        0xF, 0xF, 0xF, 0xF, 0xF
};

const unsigned char PRODUCT_KEY_PART2[] = {
        0xE, 0xE, 0xE, 0xE, 0xE,
        0xF, 0xF, 0xF, 0xF, 0xF,
        0xB, 0xB, 0xB, 0xB, 0xB,
        0xF, 0xF, 0xF, 0xF, 0xF,
        0xA, 0xA, 0xA, 0xA, 0xA
};

#define DIGITS_COUNT (GROUPS_DIGITS_COUNT * GROUP_DIGITS_SIZE)


#if _WIN32

#define TYPER_FULL_RAW_MODE

#define IS_KEY_UP(CH0, CH1, CH2) (CH0 == 224 && CH1 == 72)
#define IS_KEY_DOWN(CH0, CH1, CH2) (CH0 == 224 && CH1 == 80)
#define IS_KEY_LEFT(CH0, CH1, CH2) (CH0 == 224 && CH2 == 75)
#define IS_KEY_RIGHT(CH0, CH1, CH2) (CH0 == 224 && CH2 == 77)
#define ESCAPE_SEQUENSE_INIT(CH0, CH1) { if (CH0 == 224) CH1 = _getch(); }
#define IS_ESCAPE_SEQUENSE_PREPARE(CH0, CH1, CH2)
#define IS_ESCAPE_KEY(CH0, CH1) (CH0 == 224)
#define IS_KEY_DELETE_PREPARE(CH0, CH1, CH2, CH3)
#define IS_KEY_DELETE(CH0, CH1, CH2, CH3) (CH0 == 224 && CH1 == 83)
#define IS_KEY_BACKSPACE(CH0) (CH0 == 8)
#define IS_KEY_ENTER(CH0) (CH0 == 13)
#define IS_KEY_CTRLC(CH0) (CH0 == 3)

#else // #elif __linux__

#define TYPER_FULL_RAW_MODE

#define IS_KEY_UP(CH0, CH1, CH2) (CH0 == 0x1b && CH1 == '[' && CH2 == 'A')
#define IS_KEY_DOWN(CH0, CH1, CH2) (CH0 == 0x1b && CH1 == '[' && CH2 == 'B')
#define IS_KEY_LEFT(CH0, CH1, CH2) (CH0 == 0x1b && CH1 == '[' && CH2 == 'D')
#define IS_KEY_RIGHT(CH0, CH1, CH2) (CH0 == 0x1b && CH1 == '[' && CH2 == 'C')
#define ESCAPE_SEQUENSE_INIT(CH0, CH1) { if (!kbhit()) CH1 = 0x1b; else if (CH0 == 0x1b) CH1 = getch(); }
#define IS_ESCAPE_SEQUENSE_PREPARE(CH0, CH1, CH2) {if(CH0 == 0x1b && CH1 == '[') CH2 = getch();}
#define IS_ESCAPE_KEY(CH0, CH1) (CH0 == 0x1b && CH1 == 0x1b)
#define IS_KEY_DELETE_PREPARE(CH0, CH1, CH2, CH3) {if(CH0 == 0x1b && CH1 == '[' && CH2 == '3') CH3 = getch();}
#define IS_KEY_DELETE(CH0, CH1, CH2, CH3) (CH0 == 0x1b && CH1 == '[' && CH2 == '3') // && CH3 == '^')
#define IS_KEY_BACKSPACE(CH0) (CH0 == 127)
#ifdef TYPER_FULL_RAW_MODE
#define IS_KEY_ENTER(CH0) (CH0 == 13)
#else
#define IS_KEY_ENTER(CH0) (CH0 == 10)
#endif
```

```
#define IS_KEY_CTRLC(CH0) (CH0 == 3)

#endif

int outOfEdgeIndex = 0;
int currIndex = 0;
unsigned char data[DIGITS_COUNT] = { 0 };

char checkProductKey(unsigned char * productKey){
	unsigned int index;
	for (index = 0; index < DIGITS_COUNT; ++index){
		if (productKey[index] ^ PRODUCT_KEY_PART1[index] ^ PRODUCT_KEY_PART2[index]){
			return 0;
		}
	}

	return ~0;
}

void toDigitPosition(unsigned int currIndex){
	int positionAddon;
#if _WIN32
#else // #elif __linux__
	char temp[16];
#endif
#if _WIN32
	CONSOLE_SCREEN_BUFFER_INFO cbsi;
	COORD pos;
	HANDLE hConsoleOutput = GetStdHandle(STD_OUTPUT_HANDLE);
	GetConsoleScreenBufferInfo(hConsoleOutput, &cbsi);
	pos = cbsi.dwCursorPosition;
#endif
	positionAddon = currIndex / GROUP_DIGITS_SIZE;
	positionAddon && positionAddon >= GROUPS_DIGITS_COUNT ? --positionAddon : 0;
#if _WIN32
	currIndex += positionAddon;
	pos.X = currIndex;
	SetConsoleCursorPosition(hConsoleOutput, pos);
#else // #elif __linux__
	write(STDOUT_FILENO, "\033[64D", 5);
	if (currIndex += positionAddon){
		sprintf(temp, "\033[%dC", currIndex);
		write(STDOUT_FILENO, temp, strlen(temp));
	}
#endif
}

void printProductKey(unsigned char * productKey, unsigned int outOfEdgeIndex){
	unsigned int index;
	unsigned char value;
	for (index = 0; index < DIGITS_COUNT && index < outOfEdgeIndex; ++index){
		value = productKey[index];
		value > 9 ? (value += 'A' - 10) : (value += '0');
#if _WIN32
		printf("%c", value);
#else // #elif __linux__
		write(STDOUT_FILENO, &value, 1);
#endif
	}
}

void printFormattedProductKey(unsigned char * productKey, unsigned int outOfEdgeIndex){
	unsigned int index;
	unsigned char value;
	for (index = 0; index < DIGITS_COUNT && index < outOfEdgeIndex; ++index){
		value = productKey[index];
		value > 9 ? (value += 'A' - 10) : (value += '0');
#if _WIN32
		printf("%c", value);
#else // #elif __linux__
		write(STDOUT_FILENO, &value, 1);
#endif
		if (!((index + 1) % GROUP_DIGITS_SIZE) && (index + 1) < DIGITS_COUNT){
#if _WIN32
			printf("-");
#else // #elif __linux__
			write(STDOUT_FILENO, "-", 1);
#endif
		}
	}
}
```

```
#if _WIN32
#else // #elif __linux__

static struct termios term, oterm;

static int getch(void){
	int c = 0;

	tcgetattr(0, &oterm);
	memcpy(&term, &oterm, sizeof(term));
#ifdef TYPER_FULL_RAW_MODE
	term.c_iflag |= IGNBRK;
	term.c_iflag &= ~(INLCR | ICRNL | IXON | IXOFF);
	term.c_lflag &= ~(ICANON | ECHO | ECHOK | ECHOE | ECHONL | ISIG | IEXTEN);
#else
	term.c_lflag &= ~(ICANON | ECHO);
#endif
	term.c_cc[VMIN] = 1;
	term.c_cc[VTIME] = 0;
	tcsetattr(0, TCSANOW, &term);
	c = getchar();
	tcsetattr(0, TCSANOW, &oterm);
	return c;
}

static int kbhit(void){
	int c = 0;

	tcgetattr(0, &oterm);
	memcpy(&term, &oterm, sizeof(term));
#ifndef TYPER_FULL_RAW_MODE
	term.c_iflag |= IGNBRK;
	term.c_iflag &= ~(INLCR | ICRNL | IXON | IXOFF);
	term.c_lflag &= ~(ICANON | ECHO | ECHOK | ECHOE | ECHONL | ISIG | IEXTEN);
#else
	term.c_lflag &= ~(ICANON | ECHO);
#endif
	term.c_cc[VMIN] = 0;
	term.c_cc[VTIME] = 1;
	tcsetattr(0, TCSANOW, &term);
	c = getchar();
	tcsetattr(0, TCSANOW, &oterm);
	if (c != -1) ungetc(c, stdin);
	return ((c != -1) ? 1 : 0);
}
#endif

void stdinInputHandler(char * ch0, char * ch1, char * ch2, char * ch3) {
#if _WIN32
	*ch0 = _getch();
#else // #elif __linux__
	*ch0 = getch();
#endif
	*ch1 = 0; //
	ESCAPE_SEQUENSE_INIT(*ch0, *ch1);
	IS_ESCAPE_SEQUENSE_PREPARE(*ch0, *ch1, *ch2);
	IS_KEY_DELETE_PREPARE(*ch0, *ch1, *ch2, *ch3);
	if (IS_KEY_CTRLC(*ch0)) {
		// no action
	}
	else if (IS_KEY_ENTER(*ch0)) {
		// no action
	}
	else if (IS_KEY_BACKSPACE(*ch0)) {
		// no action
	}
	else if (IS_KEY_DELETE(*ch0, *ch1, *ch2, *ch3)) {
		// no action
	}
	else if (IS_KEY_LEFT(*ch0, *ch1, *ch2)) {
		// no action
	}
	else if (IS_KEY_RIGHT(*ch0, *ch1, *ch2)) {
		// no action
	}
	else if (IS_ESCAPE_KEY(*ch0, *ch1)){
#if _WIN32
			_getch();
#else // #elif __linux__
		while (kbhit()) {
			getch();
		}
```

```
#endif
		}
}

void inputHandler(char ch0, char ch1, char ch2, char ch3){
		char chstr_[2] = { 0 };
		char * hexDigitScanfPattern = (char*)"%[0-9abcdefABCDEF]"; // /[0-9A-Fa-f]/g

		if (!attemptsDownCount){
			return;
		}
		if (IS_KEY_ENTER(ch0)) {
			if (checkProductKey(data)) {
#if _WIN32
				printf("\nThe product key is correct\n\n");
#else // #elif __linux__
				write(STDOUT_FILENO, "\nThe product key is correct\n\n", 29);
#endif
				printProductKey(data, outOfEdgeIndex);
#if _WIN32
				printf(" (COMPLETE)", 11);
				printf("\nFor exit press Ctrl + C\n");
#else // #elif __linux__
				write(STDOUT_FILENO, " (COMPLETE)", 11);
				write(STDOUT_FILENO, "\nFor exit press Ctrl + C\n", 25);
#endif
				attemptsDownCount = 0;
			}
			else{
#if _WIN32
				printf("\nThe product key is not correct\n");
				printf("\nYou have %d attempts to try\n", --attemptsDownCount);
#else // #elif __linux__
				write(STDOUT_FILENO, "\nThe product key is not correct\n", 32);
				printf("\nYou have %d attempts to try\n", --attemptsDownCount);
#endif
				if (attemptsDownCount){
#if _WIN32
					printf("Please, enter the product key:\n");
#else // #elif __linux__
					write(STDOUT_FILENO, "Please, enter the product key:\n", 31);
#endif
					printFormattedProductKey(data, outOfEdgeIndex);
					toDigitPosition(currIndex);
				}
				else{
#if _WIN32
					printf("The product key is not entered\n");
					printf("For exit press Ctrl + C\n");
#else // #elif __linux__
					write(STDOUT_FILENO, "The product key is not entered\n", 31);
					write(STDOUT_FILENO, "For exit press Ctrl + C\n", 24);
#endif
				}
			}
		}
		else if (IS_KEY_BACKSPACE(ch0)) {
			if (currIndex){
				--currIndex;
				toDigitPosition(currIndex);
				data[currIndex] = 0;
#if _WIN32
				printf("0");
#else // #elif __linux__
				write(STDOUT_FILENO, "0", 1);
#endif
				toDigitPosition(currIndex);
			}
		}
		else if (IS_KEY_DELETE(ch0, ch1, ch2, ch3)) {
			toDigitPosition(currIndex);
			data[currIndex] = 0;
#if _WIN32
			printf("0");
#else // #elif __linux__
			write(STDOUT_FILENO, "0", 1);
#endif
			toDigitPosition(currIndex);
		}
		else if (IS_KEY_LEFT(ch0, ch1, ch2)) {
			if (currIndex){
				toDigitPosition(--currIndex); // got to 1.5
```

```
                }
        }
        else if (IS_KEY_RIGHT(ch0, ch1, ch2)) {
                if (currIndex < outOfEdgeIndex){
                        toDigitPosition(++currIndex);
                }
        }
        else if (IS_ESCAPE_KEY(ch0, ch1)){
                // no action
        }

        ch0 == ' ' || ch0 == '\t' ? ch0 = '0' : 0;

        //char chstr_[2] = { 0 };
        //char * hexDigitScanfPattern = (char*)"%[0-9abcdefABCDEF]"; // /[0-9A-Fa-f]/g

        if (currIndex < DIGITS_COUNT && ch0 && sscanf((char*)&ch0, hexDigitScanfPattern, chstr_) > 0) {
                data[currIndex] = (unsigned char)strtol(chstr_, NULL, 16);
#if _WIN32
                printf("%X", data[currIndex]);
#else // #elif __linux__
                sprintf(chstr_, "%X", data[currIndex]);
                write(STDOUT_FILENO, chstr_, 1);
#endif
                if (outOfEdgeIndex <= currIndex){
                        outOfEdgeIndex = currIndex + 1;
                }
                if (currIndex + 1 < DIGITS_COUNT) {
                        ++currIndex;
                        if (currIndex != DIGITS_COUNT && !(currIndex % 5)) {
#if _WIN32
                                printf("-");
#else // #elif __linux__
                                write(STDOUT_FILENO, "-", 1);
#endif
                        }
                }
                if (currIndex + 1 == DIGITS_COUNT){
                        toDigitPosition(currIndex);
                }
        }
}


#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
#include <functional>

class ConsoleKeyInputDataSource {
public:
        class iterator : public std::iterator<std::input_iterator_tag, int, int, const int*, int>{
                int value;// = _;
        public:
                explicit iterator(int _value = 0) : value(_value) {
                }
                iterator& operator++() {
                        stdinInputHandler((char*)&value, (char*)&value + 1, (char*)&value + 2, (char*)&value + 3);
                        return *this;
                }
                iterator operator++(int) {
                        iterator retval = *this;
                        ++(*this);
                        return retval;
                }
                bool operator==(iterator other) const {
                        return value == other.value;
                }
                bool operator!=(iterator other) const {
                        return !(*this == other);
                }
                reference operator*() const {
                        return value;
                }
        };
        iterator begin() {
                return iterator('_');
        }
        iterator end() {
#if _WIN32
                return iterator(0x3); // TODO: ...
```

```
#else // #elif __linux__
		return iterator(0x1b03); // TODO: ...
#endif
	}
};

class PrintObserver : public rxcpp::observer<int> {
public:
	void on_next(int && v) const {
		inputHandler(*(char*)&v, *((char*)&v + 1), *((char*)&v + 2), *((char*)&v + 3));
	}
	void on_error(std::exception_ptr e) const {
		try {
			if (e) {
				std::rethrow_exception(e);
			}
		}
		catch (const std::exception& exception) {
			std::cout << std::endl << "Error Occurred: \"" << exception.what() << "\"" << std::endl;
		}
	}
	void on_completed() const {
		printf("\nDone!");
	}
};

int main(){
	if (attemptsDownCount){
		printf("Please, enter the product key:\n");
	}

	auto inputObservable = rxcpp::observable<>::iterate(ConsoleKeyInputDataSource());
	//filter is not used
	//inputObservable.filter([](int v){return v == 'Q'; });
	inputObservable.map([](int v){return v == (int)' ' || v == (int)'\t' ? (int)' ' : v; });
	inputObservable.subscribe(PrintObserver());
	return 0;
}


/************************************************** RxCpp Implementation
**************************************************/
/*                                  https://github.com/ReactiveX/RxCpp)                                  */
/************************************************************************************************************************
**************/
// Copyright (c) Microsoft Open Technologies, Inc. All rights reserved. See License.txt in the project root for license information.

#else

#if !defined(RXCPP_RX_INCLUDES_HPP)
#define RXCPP_RX_INCLUDES_HPP

#if !defined(RXCPP_RX_TRACE_HPP)
#define RXCPP_RX_TRACE_HPP

#include <iostream>
#include <exception>
#include <atomic>

namespace rxcpp {

	struct trace_id
	{
		static inline trace_id make_next_id_subscriber() {
			static std::atomic<unsigned long> id(0xB0000000);
			return trace_id{ ++id };
		}
		unsigned long id;
	};

	inline bool operator==(const trace_id& lhs, const trace_id& rhs) {
		return lhs.id == rhs.id;
	}
	inline bool operator!=(const trace_id& lhs, const trace_id& rhs) {
		return !(lhs == rhs);
	}

	inline bool operator<(const trace_id& lhs, const trace_id& rhs) {
		if ((lhs.id & 0xF0000000) != (rhs.id & 0xF0000000)) std::terminate();
		return lhs.id < rhs.id;
	}
	inline bool operator>(const trace_id& lhs, const trace_id& rhs) {
```

```
            return rhs<lhs;
        }

        inline std::ostream& operator<< (std::ostream& os, const trace_id& id) {
            return os << std::hex << id.id << std::dec;
        }

        struct trace_noop
        {
            template<class Worker, class Schedulable>
            inline void schedule_enter(const Worker&, const Schedulable&) {}
            template<class Worker>
            inline void schedule_return(const Worker&) {}
            template<class Worker, class When, class Schedulable>
            inline void schedule_when_enter(const Worker&, const When&, const Schedulable&) {}
            template<class Worker>
            inline void schedule_when_return(const Worker&) {}

            template<class Schedulable>
            inline void action_enter(const Schedulable&) {}
            template<class Schedulable>
            inline void action_return(const Schedulable&) {}
            template<class Schedulable>
            inline void action_recurse(const Schedulable&) {}

            template<class Observable, class Subscriber>
            inline void subscribe_enter(const Observable&, const Subscriber&) {}
            template<class Observable>
            inline void subscribe_return(const Observable&) {}

            template<class SubscriberFrom, class SubscriberTo>
            inline void connect(const SubscriberFrom&, const SubscriberTo&) {}

            template<class OperatorSource, class OperatorChain, class Subscriber, class SubscriberLifted>
            inline void lift_enter(const OperatorSource&, const OperatorChain&, const Subscriber&, const SubscriberLifted&) {}
            template<class OperatorSource, class OperatorChain>
            inline void lift_return(const OperatorSource&, const OperatorChain&) {}

            template<class SubscriptionState>
            inline void unsubscribe_enter(const SubscriptionState&) {}
            template<class SubscriptionState>
            inline void unsubscribe_return(const SubscriptionState&) {}

            template<class SubscriptionState, class Subscription>
            inline void subscription_add_enter(const SubscriptionState&, const Subscription&) {}
            template<class SubscriptionState>
            inline void subscription_add_return(const SubscriptionState&) {}

            template<class SubscriptionState, class WeakSubscription>
            inline void subscription_remove_enter(const SubscriptionState&, const WeakSubscription&) {}
            template<class SubscriptionState>
            inline void subscription_remove_return(const SubscriptionState&) {}

            template<class Subscriber>
            inline void create_subscriber(const Subscriber&) {}

            template<class Subscriber, class T>
            inline void on_next_enter(const Subscriber&, const T&) {}
            template<class Subscriber>
            inline void on_next_return(const Subscriber&) {}

            template<class Subscriber>
            inline void on_error_enter(const Subscriber&, const std::exception_ptr&) {}
            template<class Subscriber>
            inline void on_error_return(const Subscriber&) {}

            template<class Subscriber>
            inline void on_completed_enter(const Subscriber&) {}
            template<class Subscriber>
            inline void on_completed_return(const Subscriber&) {}
        };

        struct trace_tag {};

}

inline auto rxcpp_trace_activity(...)->rxcpp::trace_noop;


#endif

// some configuration macros
```

```
#if defined(_MSC_VER)

#if _MSC_VER > 1600
#pragma warning(disable: 4348) // false positives on : redefinition of default parameter : parameter 2
#define RXCPP_USE_RVALUEREF 1
#endif

#if _MSC_VER >= 1800
#define RXCPP_USE_VARIADIC_TEMPLATES 1
#endif

#if _CPPRTTI
#define RXCPP_USE_RTTI 1
#endif

#elif defined(__clang__)

#if __has_feature(cxx_rvalue_references)
#define RXCPP_USE_RVALUEREF 1
#endif
#if __has_feature(cxx_rtti)
#define RXCPP_USE_RTTI 1
#endif
#if __has_feature(cxx_variadic_templates)
#define RXCPP_USE_VARIADIC_TEMPLATES 1
#endif

#elif defined(__GNUG__)

#define GCC_VERSION (__GNUC__ * 10000 + \
                     __GNUC_MINOR__ * 100 + \
                     __GNUC_PATCHLEVEL__)

#if GCC_VERSION >= 40801
#define RXCPP_USE_RVALUEREF 1
#endif

#if GCC_VERSION >= 40400
#define RXCPP_USE_VARIADIC_TEMPLATES 1
#endif

#if defined(__GXX_RTTI)
#define RXCPP_USE_RTTI 1
#endif

#endif

//
// control std::hash<> of enum
// force with RXCPP_FORCE_HASH_ENUM & RXCPP_FORCE_HASH_ENUM_UNDERLYING
// in time use ifdef to detect library support for std::hash<> of enum
//
#define RXCPP_HASH_ENUM 0
#define RXCPP_HASH_ENUM_UNDERLYING 1

#if !defined(WINAPI_FAMILY) || (WINAPI_FAMILY == WINAPI_FAMILY_DESKTOP_APP)
#define RXCPP_USE_WINRT 0
#else
#define RXCPP_USE_WINRT 1
#endif

#if defined(__APPLE__) && defined(__MACH__)
#include <TargetConditionals.h>
#if (TARGET_OS_IPHONE == 1) || (TARGET_IPHONE_SIMULATOR == 1)
#define RXCPP_ON_IOS
#endif
#endif

#if defined(__ANDROID__)
#define RXCPP_ON_ANDROID
#endif

#if defined(RXCPP_FORCE_USE_VARIADIC_TEMPLATES)
#undef RXCPP_USE_VARIADIC_TEMPLATES
#define RXCPP_USE_VARIADIC_TEMPLATES RXCPP_FORCE_USE_VARIADIC_TEMPLATES
#endif

#if defined(RXCPP_FORCE_USE_RVALUEREF)
#undef RXCPP_USE_RVALUEREF
#define RXCPP_USE_RVALUEREF RXCPP_FORCE_USE_RVALUEREF
#endif
```

```
#if defined(RXCPP_FORCE_USE_RTTI)
#undef RXCPP_USE_RTTI
#define RXCPP_USE_RTTI RXCPP_FORCE_USE_RTTI
#endif

#if defined(RXCPP_FORCE_USE_WINRT)
#undef RXCPP_USE_WINRT
#define RXCPP_USE_WINRT RXCPP_FORCE_USE_WINRT
#endif

#if defined(RXCPP_FORCE_HASH_ENUM)
#undef RXCPP_HASH_ENUM
#define RXCPP_HASH_ENUM RXCPP_FORCE_HASH_ENUM
#endif

#if defined(RXCPP_FORCE_HASH_ENUM_UNDERLYING)
#undef RXCPP_HASH_ENUM_UNDERLYING
#define RXCPP_HASH_ENUM_UNDERLYING RXCPP_FORCE_HASH_ENUM_UNDERLYING
#endif

#if defined(RXCPP_FORCE_ON_IOS)
#undef RXCPP_ON_IOS
#define RXCPP_ON_IOS RXCPP_FORCE_ON_IOS
#endif

#if defined(RXCPP_FORCE_ON_ANDROID)
#undef RXCPP_ON_ANDROID
#define RXCPP_ON_ANDROID RXCPP_FORCE_ON_ANDROID
#endif

#if defined(_MSC_VER) && !RXCPP_USE_VARIADIC_TEMPLATES
// resolve args needs enough to store all the possible resolved args
#define _VARIADIC_MAX 10
#endif

#pragma push_macro("min")
#pragma push_macro("max")
#undef min
#undef max

#include <stdlib.h>

#include <cstddef>

#include <iostream>
#include <iomanip>

#include <exception>
#include <functional>
#include <memory>
#include <array>
#include <vector>
#include <algorithm>
#include <atomic>
#include <map>
#include <set>
#include <mutex>
#include <deque>
#include <thread>
#include <future>
#include <vector>
#include <list>
#include <queue>
#include <chrono>
#include <condition_variable>
#include <initializer_list>
#include <typeinfo>
#include <tuple>
#include <unordered_set>
#include <type_traits>
#include <utility>

#if defined(RXCPP_ON_IOS) || defined(RXCPP_ON_ANDROID)
#include <pthread.h>
#endif

//#pragma once

#if !defined(RXCPP_RX_UTIL_HPP)
#define RXCPP_RX_UTIL_HPP

// _include "rx-includes.hpp"
```

```
#if !defined(RXCPP_ON_IOS) && !defined(RXCPP_ON_ANDROID) && !defined(RXCPP_THREAD_LOCAL)
#if defined(_MSC_VER)
#define RXCPP_THREAD_LOCAL __declspec(thread)
#else
#define RXCPP_THREAD_LOCAL __thread
#endif
#endif

#if !defined(RXCPP_DELETE)
#if defined(_MSC_VER)
#define RXCPP_DELETE __pragma(warning(disable: 4822)) =delete
#else
#define RXCPP_DELETE =delete
#endif
#endif

#define RXCPP_CONCAT(Prefix, Suffix) Prefix ## Suffix
#define RXCPP_CONCAT_EVALUATE(Prefix, Suffix) RXCPP_CONCAT(Prefix, Suffix)

#define RXCPP_MAKE_IDENTIFIER(Prefix) RXCPP_CONCAT_EVALUATE(Prefix, __LINE__)

namespace rxcpp {

	namespace util {

		template<class T> using value_type_t = typename std::decay<T>::type::value_type;
		template<class T> using decay_t = typename std::decay<T>::type;
		template<class... TN> using result_of_t = typename std::result_of<TN...>::type;

		template<class T, std::size_t size>
		std::vector<T> to_vector(const T(&arr)[size]) {
			return std::vector<T>(std::begin(arr), std::end(arr));
		}

		template<class T>
		std::vector<T> to_vector(std::initializer_list<T> il) {
			return std::vector<T>(il);
		}

		template<class T0, class... TN>
		typename std::enable_if<!std::is_array<T0>::value && std::is_pod<T0>::value, std::vector<T0>>::type to_vector(T0 t0,
TN... tn) {
			return to_vector({ t0, tn... });
		}

		template<class T, T... ValueN>
		struct values {};

		template<class T, int Remaining, T Step = 1, T Cursor = 0, T... ValueN>
		struct values_from;

		template<class T, T Step, T Cursor, T... ValueN>
		struct values_from<T, 0, Step, Cursor, ValueN...>
		{
			typedef values<T, ValueN...> type;
		};

		template<class T, int Remaining, T Step, T Cursor, T... ValueN>
		struct values_from
		{
			typedef typename values_from<T, Remaining - 1, Step, Cursor + Step, ValueN..., Cursor>::type type;
		};

		template<bool... BN>
		struct all_true;

		template<bool B>
		struct all_true<B>
		{
			static const bool value = B;
		};
		template<bool B, bool... BN>
		struct all_true<B, BN...>
		{
			static const bool value = B && all_true<BN...>::value;
		};

		template<bool... BN>
		using enable_if_all_true_t = typename std::enable_if<all_true<BN...>::value>::type;

		template<class... BN>
```

```
struct all_true_type;

template<class B>
struct all_true_type<B>
{
	static const bool value = B::value;
};
template<class B, class... BN>
struct all_true_type<B, BN...>
{
	static const bool value = B::value && all_true_type<BN...>::value;
};

template<class... BN>
using enable_if_all_true_type_t = typename std::enable_if<all_true_type<BN...>::value>::type;

struct all_values_true {
	template<class... ValueN>
	bool operator()(ValueN... vn) const;

	template<class Value0>
	bool operator()(Value0 v0) const {
		return v0;
	}

	template<class Value0, class... ValueN>
	bool operator()(Value0 v0, ValueN... vn) const {
		return v0 && all_values_true()(vn...);
	}
};

struct any_value_true {
	template<class... ValueN>
	bool operator()(ValueN... vn) const;

	template<class Value0>
	bool operator()(Value0 v0) const {
		return v0;
	}

	template<class Value0, class... ValueN>
	bool operator()(Value0 v0, ValueN... vn) const {
		return v0 || all_values_true()(vn...);
	}
};

template<class... TN>
struct types {};

//
// based on Walter Brown's void_t proposal
// http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3911.pdf
//

struct types_checked {};

namespace detail {
	template<class... TN> struct types_checked_from { typedef types_checked type; };
}

template<class... TN>
struct types_checked_from { typedef typename detail::types_checked_from<TN...>::type type; };

template<class... TN>
using types_checked_t = typename types_checked_from<TN...>::type;


template<class Types, class = types_checked>
struct expand_value_types { struct type; };
template<class... TN>
struct expand_value_types<types<TN...>, types_checked_t<typename std::decay<TN>::type::value_type...>>
{
	using type = types<typename std::decay<TN>::type::value_type...>;
};
template<class... TN>
using value_types_t = typename expand_value_types<types<TN...>>::type;


template<class T, class C = types_checked>
struct value_type_from : public std::false_type { typedef types_checked type; };

template<class T>
```

```
struct value_type_from<T, typename types_checked_from<value_type_t<T>>::type>
	: public std::true_type{ typedef value_type_t<T> type; };

namespace detail {
	template<class F, class... ParamN, int... IndexN>
	auto apply(std::tuple<ParamN...> p, values<int, IndexN...>, F&& f)
		-> decltype(f(std::forward<ParamN>(std::get<IndexN>(p))...)) {
		return	f(std::forward<ParamN>(std::get<IndexN>(p))...);
	}

	template<class F_inner, class F_outer, class... ParamN, int... IndexN>
	auto apply_to_each(std::tuple<ParamN...>& p, values<int, IndexN...>, F_inner& f_inner, F_outer& f_outer)
		-> decltype(f_outer(std::move(f_inner(std::get<IndexN>(p))))...)) {
		return	f_outer(std::move(f_inner(std::get<IndexN>(p))))...);
	}

	template<class F_inner, class F_outer, class... ParamN, int... IndexN>
	auto apply_to_each(std::tuple<ParamN...>& p, values<int, IndexN...>, const F_inner& f_inner, const F_outer&
f_outer)
		-> decltype(f_outer(std::move(f_inner(std::get<IndexN>(p))))...)) {
		return	f_outer(std::move(f_inner(std::get<IndexN>(p))))...);
	}

}
template<class F, class... ParamN>
auto apply(std::tuple<ParamN...> p, F&& f)
	-> decltype(detail::apply(std::move(p), typename values_from<int, sizeof...(ParamN)>::type(),
std::forward<F>(f))) {
	return	detail::apply(std::move(p), typename values_from<int, sizeof...(ParamN)>::type(), std::forward<F>(f));
}

template<class F_inner, class F_outer, class... ParamN>
auto apply_to_each(std::tuple<ParamN...>& p, F_inner& f_inner, F_outer& f_outer)
	-> decltype(detail::apply_to_each(p, typename values_from<int, sizeof...(ParamN)>::type(), f_inner, f_outer)) {
	return	detail::apply_to_each(p, typename values_from<int, sizeof...(ParamN)>::type(), f_inner, f_outer);
}

template<class F_inner, class F_outer, class... ParamN>
auto apply_to_each(std::tuple<ParamN...>& p, const F_inner& f_inner, const F_outer& f_outer)
	-> decltype(detail::apply_to_each(p, typename values_from<int, sizeof...(ParamN)>::type(), f_inner, f_outer)) {
	return	detail::apply_to_each(p, typename values_from<int, sizeof...(ParamN)>::type(), f_inner, f_outer);
}

namespace detail {

	template<class F>
	struct apply_to
	{
		F to;

		explicit apply_to(F f)
			: to(std::move(f))
		{
		}

		template<class... ParamN>
		auto operator()(std::tuple<ParamN...> p)
			-> decltype(rxcpp::util::apply(std::move(p), to)) {
			return	rxcpp::util::apply(std::move(p), to);
		}
		template<class... ParamN>
		auto operator()(std::tuple<ParamN...> p) const
			-> decltype(rxcpp::util::apply(std::move(p), to)) {
			return	rxcpp::util::apply(std::move(p), to);
		}
	};

}

template<class F>
auto apply_to(F f)
	->	detail::apply_to<F> {
	return	detail::apply_to<F>(std::move(f));
}

namespace detail {

	struct pack
	{
		template<class... ParamN>
		auto operator()(ParamN... pn)
			-> decltype(std::make_tuple(std::move(pn)...)) {
```

```
                                        return      std::make_tuple(std::move(pn)...);
                                }
                                template<class... ParamN>
                                auto operator()(ParamN... pn) const
                                        -> decltype(std::make_tuple(std::move(pn)...)) {
                                        return      std::make_tuple(std::move(pn)...);
                                }
                        };

        }

        inline auto pack()
                ->      detail::pack {
                return  detail::pack();
        }

        namespace detail {

                template<int Index>
                struct take_at
                {
                        template<class... ParamN>
                        auto operator()(ParamN... pn)
                                -> typename std::tuple_element<Index, std::tuple<decay_t<ParamN>...>>::type {
                                return              std::get<Index>(std::make_tuple(std::move(pn)...));
                        }
                        template<class... ParamN>
                        auto operator()(ParamN... pn) const
                                -> typename std::tuple_element<Index, std::tuple<decay_t<ParamN>...>>::type {
                                return              std::get<Index>(std::make_tuple(std::move(pn)...));
                        }
                };

        }

        template<int Index>
        inline auto take_at()
                ->      detail::take_at<Index> {
                return  detail::take_at<Index>();
        }

        template <class D>
        struct resolve_type;

        template <template<class... TN> class Deferred, class... AN>
        struct defer_trait
        {
                template<bool R>
                struct tag_valid { static const bool valid = true; static const bool value = R; };
                struct tag_not_valid { static const bool valid = false; static const bool value = false; };
                typedef Deferred<typename resolve_type<AN>::type...> resolved_type;
                template<class... CN>
                static auto check(int)->tag_valid<resolved_type::value>;
                template<class... CN>
                static tag_not_valid check(...);

                typedef decltype(check<AN...>(0)) tag_type;
                static const bool valid = tag_type::valid;
                static const bool value = tag_type::value;
                static const bool not_value = valid && !value;
        };

        template <template<class... TN> class Deferred, class... AN>
        struct defer_type
        {
                template<class R>
                struct tag_valid { typedef R type; static const bool value = true; };
                struct tag_not_valid { typedef void type; static const bool value = false; };
                typedef Deferred<typename resolve_type<AN>::type...> resolved_type;
                template<class... CN>
                static auto check(int)->tag_valid<resolved_type>;
                template<class... CN>
                static tag_not_valid check(...);

                typedef decltype(check<AN...>(0)) tag_type;
                typedef typename tag_type::type type;
                static const bool value = tag_type::value;
        };

        template <template<class... TN> class Deferred, class... AN>
        struct defer_value_type
        {
```

```
		template<class R>
		struct tag_valid { typedef R type; static const bool value = true; };
		struct tag_not_valid { typedef void type; static const bool value = false; };
		typedef Deferred<typename resolve_type<AN>::type...> resolved_type;
		template<class... CN>
		static auto check(int)->tag_valid<value_type_t<resolved_type>>;
		template<class... CN>
		static tag_not_valid check(...);

		typedef decltype(check<AN...>(0)) tag_type;
		typedef typename tag_type::type type;
		static const bool value = tag_type::value;
};

template <template<class... TN> class Deferred, class... AN>
struct defer_seed_type
{
		template<class R>
		struct tag_valid { typedef R type; static const bool value = true; };
		struct tag_not_valid { typedef void type; static const bool value = false; };
		typedef Deferred<typename resolve_type<AN>::type...> resolved_type;
		template<class... CN>
		static auto check(int)->tag_valid<typename resolved_type::seed_type>;
		template<class... CN>
		static tag_not_valid check(...);

		typedef decltype(check<AN...>(0)) tag_type;
		typedef typename tag_type::type type;
		static const bool value = tag_type::value;
};

template <class D>
struct resolve_type
{
		typedef D type;
};
template <template<class... TN> class Deferred, class... AN>
struct resolve_type<defer_type<Deferred, AN...>>
{
		typedef typename defer_type<Deferred, AN...>::type type;
};
template <template<class... TN> class Deferred, class... AN>
struct resolve_type<defer_value_type<Deferred, AN...>>
{
		typedef typename defer_value_type<Deferred, AN...>::type type;
};
template <template<class... TN> class Deferred, class... AN>
struct resolve_type<defer_seed_type<Deferred, AN...>>
{
		typedef typename defer_seed_type<Deferred, AN...>::type type;
};

struct plus
{
		template <class LHS, class RHS>
		auto operator()(LHS&& lhs, RHS&& rhs) const
			-> decltype(std::forward<LHS>(lhs) +std::forward<RHS>(rhs))
		{
			return std::forward<LHS>(lhs) +std::forward<RHS>(rhs);
		}
};

struct count
{
		template <class T>
		int operator()(int cnt, T&&) const
		{
			return cnt + 1;
		}
};

struct less
{
		template <class LHS, class RHS>
		auto operator()(LHS&& lhs, RHS&& rhs) const
			-> decltype(std::forward<LHS>(lhs) < std::forward<RHS>(rhs))
		{
			return std::forward<LHS>(lhs) < std::forward<RHS>(rhs);
		}
};

template<class T = void>
```

```
struct equal_to
{
	bool operator()(const T& lhs, const T& rhs) const { return lhs == rhs; }
};

template<>
struct equal_to<void>
{
	template<class LHS, class RHS>
	auto operator()(LHS&& lhs, RHS&& rhs) const
		-> decltype(std::forward<LHS>(lhs) == std::forward<RHS>(rhs))
	{
		return std::forward<LHS>(lhs) == std::forward<RHS>(rhs);
	}
};

namespace detail {
	template<class OStream, class Delimit>
	struct print_function
	{
		OStream& os;
		Delimit delimit;
		print_function(OStream& os, Delimit d) : os(os), delimit(std::move(d)) {}

		template<class... TN>
		void operator()(const TN&... tn) const {
			bool inserts[] = { (os << tn, true)... };
			inserts[0] = *reinterpret_cast<bool*>(inserts); // silence warning
			delimit();
		}

		template<class... TN>
		void operator()(const std::tuple<TN...>& tpl) const {
			rxcpp::util::apply(tpl, *this);
		}
	};

	template<class OStream>
	struct endline
	{
		OStream& os;
		endline(OStream& os) : os(os) {}
		void operator()() const {
			os << std::endl;
		}
	private:
		endline& operator=(const endline&)RXCPP_DELETE;
	};

	template<class OStream, class ValueType>
	struct insert_value
	{
		OStream& os;
		ValueType value;
		insert_value(OStream& os, ValueType v) : os(os), value(std::move(v)) {}
		void operator()() const {
			os << value;
		}
	private:
		insert_value& operator=(const insert_value&)RXCPP_DELETE;
	};

	template<class OStream, class Function>
	struct insert_function
	{
		OStream& os;
		Function call;
		insert_function(OStream& os, Function f) : os(os), call(std::move(f)) {}
		void operator()() const {
			call(os);
		}
	private:
		insert_function& operator=(const insert_function&)RXCPP_DELETE;
	};

	template<class OStream, class Delimit>
	auto print_followed_with(OStream& os, Delimit d)
		->	detail::print_function<OStream, Delimit> {
		return  detail::print_function<OStream, Delimit>(os, std::move(d));
	}

}
```

```
template<class OStream>
auto endline(OStream& os)
	-> detail::endline<OStream> {
	return detail::endline<OStream>(os);
}

template<class OStream>
auto println(OStream& os)
	-> decltype(detail::print_followed_with(os, endline(os))) {
	return	detail::print_followed_with(os, endline(os));
}
template<class OStream, class Delimit>
auto print_followed_with(OStream& os, Delimit d)
	-> decltype(detail::print_followed_with(os, detail::insert_function<OStream, Delimit>(os, std::move(d)))) {
	return	detail::print_followed_with(os, detail::insert_function<OStream, Delimit>(os, std::move(d)));
}
template<class OStream, class DelimitValue>
auto print_followed_by(OStream& os, DelimitValue dv)
	-> decltype(detail::print_followed_with(os, detail::insert_value<OStream, DelimitValue>(os, std::move(dv)))) {
	return	detail::print_followed_with(os, detail::insert_value<OStream, DelimitValue>(os, std::move(dv)));
}

inline std::string what(std::exception_ptr ep) {
	try { std::rethrow_exception(ep); }
	catch (const std::exception& ex) {
		return ex.what();
	}
	return std::string();
}

namespace detail {

	template <class T>
	class maybe
	{
		bool is_set;
		typename std::aligned_storage<sizeof(T), std::alignment_of<T>::value>::type
			storage;
	public:
		maybe()
			: is_set(false)
		{
		}

		maybe(T value)
			: is_set(false)
		{
			new (reinterpret_cast<T*>(&storage)) T(value);
			is_set = true;
		}

		maybe(const maybe& other)
			: is_set(false)
		{
			if (other.is_set) {
				new (reinterpret_cast<T*>(&storage)) T(other.get());
				is_set = true;
			}
		}
		maybe(maybe&& other)
			: is_set(false)
		{
			if (other.is_set) {
				new (reinterpret_cast<T*>(&storage)) T(std::move(other.get()));
				is_set = true;
				other.reset();
			}
		}

		~maybe()
		{
			reset();
		}

		typedef T value_type;
		typedef T* iterator;
		typedef const T* const_iterator;

		bool empty() const {
			return !is_set;
		}
```

```
                        std::size_t size() const {
                                return is_set ? 1 : 0;
                        }

                        iterator begin() {
                                return reinterpret_cast<T*>(&storage);
                        }
                        const_iterator begin() const {
                                return reinterpret_cast<T*>(&storage);
                        }

                        iterator end() {
                                return reinterpret_cast<T*>(&storage) + size();
                        }
                        const_iterator end() const {
                                return reinterpret_cast<T*>(&storage) + size();
                        }

                        T* operator->() {
                                if (!is_set) std::terminate();
                                return reinterpret_cast<T*>(&storage);
                        }
                        const T* operator->() const {
                                if (!is_set) std::terminate();
                                return reinterpret_cast<T*>(&storage);
                        }

                        T& operator*() {
                                if (!is_set) std::terminate();
                                return *reinterpret_cast<T*>(&storage);
                        }
                        const T& operator*() const {
                                if (!is_set) std::terminate();
                                return *reinterpret_cast<T*>(&storage);
                        }

                        T& get() {
                                if (!is_set) std::terminate();
                                return *reinterpret_cast<T*>(&storage);
                        }
                        const T& get() const {
                                if (!is_set) std::terminate();
                                return *reinterpret_cast<const T*>(&storage);
                        }

                        void reset()
                        {
                                if (is_set) {
                                        is_set = false;
                                        reinterpret_cast<T*>(&storage)->~T();
                                        //std::fill_n(reinterpret_cast<char*>(&storage), sizeof(T), 0);
                                }
                        }

                        template<class U>
                        void reset(U&& value) {
                                reset();
                                new (reinterpret_cast<T*>(&storage)) T(std::forward<U>(value));
                                is_set = true;
                        }

                        maybe& operator=(const T& other) {
                                reset(other);
                                return *this;
                        }
                        maybe& operator=(const maybe& other) {
                                if (!other.empty()) {
                                        reset(other.get());
                                }
                                else {
                                        reset();
                                }
                                return *this;
                        }
                };

        }
        using detail::maybe;

        namespace detail {
                struct surely
```

```
                {
                        template<class... T>
                        auto operator()(T... t)
                                -> decltype(std::make_tuple(t.get()...)) {
                                return  std::make_tuple(t.get()...);
                        }
                        template<class... T>
                        auto operator()(T... t) const
                                -> decltype(std::make_tuple(t.get()...)) {
                                return  std::make_tuple(t.get()...);
                        }
                };
        }

        template<class... T>
        inline auto surely(const std::tuple<T...>& tpl)
                -> decltype(apply(tpl, detail::surely())) {
                return  apply(tpl, detail::surely());
        }

        namespace detail {

                template<typename Function>
                class unwinder
                {
                public:
                        ~unwinder()
                        {
                                if (!!function)
                                {
                                        try {
                                                (*function)();
                                        }
                                        catch (...) {
                                                std::unexpected();
                                        }
                                }
                        }

                        explicit unwinder(Function* functionArg)
                                : function(functionArg)
                        {
                        }

                        void dismiss()
                        {
                                function = nullptr;
                        }

                private:
                        unwinder();
                        unwinder(const unwinder&);
                        unwinder& operator=(const unwinder&);

                        Function* function;
                };

        }

#if !defined(RXCPP_THREAD_LOCAL)
        template<typename T>
        class thread_local_storage
        {
        private:
                pthread_key_t key;

        public:
                thread_local_storage()
                {
                        pthread_key_create(&key, NULL);
                }

                ~thread_local_storage()
                {
                        pthread_key_delete(key);
                }

                thread_local_storage& operator =(T* p)
                {
                        pthread_setspecific(key, p);
                        return *this;
                }
```

```
                                bool operator !()
                                {
                                        return pthread_getspecific(key) == NULL;
                                }

                                T* operator ->()
                                {
                                        return static_cast<T*>(pthread_getspecific(key));
                                }

                                T* get()
                                {
                                        return static_cast<T*>(pthread_getspecific(key));
                                }
                        };
#endif

                        template<typename, typename C = types_checked>
                        struct is_string : std::false_type {
                        };

                        template <typename T>
                        struct is_string<T,
                                typename types_checked_from<
                                typename T::value_type,
                                typename T::traits_type,
                                typename T::allocator_type>::type>
                                : std::is_base_of<
                                std::basic_string<
                                typename T::value_type,
                                typename T::traits_type,
                                typename T::allocator_type>, T>{
                        };

                        namespace detail {

                                template <class T, class = types_checked>
                                struct is_duration : std::false_type {};

                                template <class T>
                                struct is_duration<T, types_checked_t<T, typename T::rep, typename T::period>>
                                        : std::is_convertible<T*, std::chrono::duration<typename T::rep, typename T::period>*>{};

                        }

                        template <class T, class Decayed = decay_t<T>>
                        struct is_duration : detail::is_duration<Decayed> {};

                }
                namespace rxu = util;


                //
                // due to an noisy static_assert issue in more than one std lib impl,
                // rxcpp maintains a whitelist filter for the types that are allowed
                // to be hashed. this allows is_hashable<T> to work.
                //
                // NOTE: this should eventually be removed!
                //
                template <class T, typename = void>
                struct filtered_hash;

#if RXCPP_HASH_ENUM
                template <class T>
                struct filtered_hash<T, typename std::enable_if<std::is_enum<T>::value>::type> : std::hash<T>{
                };
#elif RXCPP_HASH_ENUM_UNDERLYING
                template <class T>
                struct filtered_hash<T, typename std::enable_if<std::is_enum<T>::value>::type> : std::hash<typename
std::underlying_type<T>::type>{
                };
#endif

                template <class T>
                struct filtered_hash<T, typename std::enable_if<std::is_integral<T>::value>::type> : std::hash<T>{
                };
                template <class T>
                struct filtered_hash<T, typename std::enable_if<std::is_pointer<T>::value>::type> : std::hash<T>{
                };
                template <class T>
                struct filtered_hash<T, typename std::enable_if<rxu::is_string<T>::value>::type> : std::hash<T>{
```

```
	};
	template <class T>
	struct filtered_hash<T, typename std::enable_if<std::is_convertible<T, std::chrono::duration<typename T::rep, typename
T::period>>::value>::type> {
		using argument_type = T;
		using result_type = std::size_t;

		result_type operator()(argument_type const & dur) const
		{
			return std::hash<typename argument_type::rep>{}(dur.count());
		}
	};
	template <class T>
	struct filtered_hash<T, typename std::enable_if<std::is_convertible<T, std::chrono::time_point<typename T::clock, typename
T::duration>>::value>::type> {
		using argument_type = T;
		using result_type = std::size_t;

		result_type operator()(argument_type const & tp) const
		{
			return std::hash<typename argument_type::rep>{}(tp.time_since_epoch().count());
		}
	};

	template<typename, typename C = rxu::types_checked>
	struct is_hashable
		: std::false_type {};

	template<typename T>
	struct is_hashable<T,
		typename rxu::types_checked_from<
		typename filtered_hash<T>::result_type,
		typename filtered_hash<T>::argument_type,
		typename std::result_of<filtered_hash<T>(T)>::type>::type>
		: std::true_type{};

}

#define RXCPP_UNWIND(Name, Function) \
	RXCPP_UNWIND_EXPLICIT(uwfunc_ ## Name, Name, Function)

#define RXCPP_UNWIND_AUTO(Function) \
	RXCPP_UNWIND_EXPLICIT(RXCPP_MAKE_IDENTIFIER(uwfunc_), RXCPP_MAKE_IDENTIFIER(unwind_), Function)

#define RXCPP_UNWIND_EXPLICIT(FunctionName, UnwinderName, Function) \
	auto FunctionName = (Function); \
	rxcpp::util::detail::unwinder<decltype(FunctionName)> UnwinderName(std::addressof(FunctionName))

#endif


#if !defined(RXCPP_RX_PREDEF_HPP)
#define RXCPP_RX_PREDEF_HPP

//_include "rx-includes.hpp"

namespace rxcpp {

	//
	// create a typedef for rxcpp_trace_type to override the default
	//
	inline auto trace_activity() -> decltype(rxcpp_trace_activity(trace_tag()))& {
		static decltype(rxcpp_trace_activity(trace_tag())) trace;
		return trace;
	}


	struct tag_action {};
	template<class T, class C = rxu::types_checked>
	struct is_action : public std::false_type {};

	template<class T>
	struct is_action<T, typename rxu::types_checked_from<typename T::action_tag>::type>
		: public std::is_convertible<typename T::action_tag*, tag_action*>{};


	struct tag_worker {};
	template<class T>
	class is_worker
	{
		struct not_void {};
		template<class C>
```

```
        static typename C::worker_tag* check(int);
        template<class C>
        static not_void check(...);
public:
        static const bool value = std::is_convertible<decltype(check<rxu::decay_t<T>>(0)), tag_worker*>::value;
};

struct tag_scheduler {};
template<class T>
class is_scheduler
{
        struct not_void {};
        template<class C>
        static typename C::scheduler_tag* check(int);
        template<class C>
        static not_void check(...);
public:
        static const bool value = std::is_convertible<decltype(check<rxu::decay_t<T>>(0)), tag_scheduler*>::value;
};

struct tag_schedulable {};
template<class T>
class is_schedulable
{
        struct not_void {};
        template<class C>
        static typename C::schedulable_tag* check(int);
        template<class C>
        static not_void check(...);
public:
        static const bool value = std::is_convertible<decltype(check<rxu::decay_t<T>>(0)), tag_schedulable*>::value;
};

namespace detail
{

        struct stateless_observer_tag {};

}

// state with optional overrides
template<class T, class State = void, class OnNext = void, class OnError = void, class OnCompleted = void>
class observer;

// no state with optional overrides
template<class T, class OnNext, class OnError, class OnCompleted>
class observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>;

// virtual functions forward to dynamically allocated shared observer instance.
template<class T>
class observer<T, void, void, void, void>;

struct tag_observer {};
template<class T>
class is_observer
{
        template<class C>
        static typename C::observer_tag* check(int);
        template<class C>
        static void check(...);
public:
        static const bool value = std::is_convertible<decltype(check<rxu::decay_t<T>>(0)), tag_observer*>::value;
};

struct tag_dynamic_observer {};
template<class T>
class is_dynamic_observer
{
        struct not_void {};
        template<class C>
        static typename C::dynamic_observer_tag* check(int);
        template<class C>
        static not_void check(...);
public:
        static const bool value = std::is_convertible<decltype(check<rxu::decay_t<T>>(0)), tag_dynamic_observer*>::value;
};

struct tag_subscriber {};
template<class T>
class is_subscriber
{
        struct not_void {};
```

```
                template<class C>
                static typename C::subscriber_tag* check(int);
                template<class C>
                static not_void check(...);
        public:
                static const bool value = std::is_convertible<decltype(check<rxu::decay_t<T>>(0)), tag_subscriber*>::value;
        };

        struct tag_dynamic_observable {};
        template<class T>
        class is_dynamic_observable
        {
                struct not_void {};
                template<class C>
                static typename C::dynamic_observable_tag* check(int);
                template<class C>
                static not_void check(...);
        public:
                static const bool value = std::is_convertible<decltype(check<rxu::decay_t<T>>(0)), tag_dynamic_observable*>::value;
        };

        template<class T>
        class dynamic_observable;

        template<
                class T = void,
                class SourceObservable = typename std::conditional<std::is_same<T, void>::value,
                        void, dynamic_observable<T >> ::type>
                        class observable;

                        template<class T, class Source>
                        observable<T> make_observable_dynamic(Source&&);

                        template<class Selector, class Default, template<class... TN> class SO, class... AN>
                        struct defer_observable;

                        struct tag_observable {};
                        template<class T>
                        struct observable_base {
                                typedef tag_observable observable_tag;
                                typedef T value_type;
                        };

                        namespace detail {

                                template<class T, class = rxu::types_checked>
                                struct is_observable : std::false_type
                                {
                                };

                                template<class T>
                                struct is_observable<T, rxu::types_checked_t<typename T::observable_tag>>
                                        : std::is_convertible<typename T::observable_tag*, tag_observable*>
                                {
                                };

                        }

                        template<class T, class Decayed = rxu::decay_t<T>>
                        struct is_observable : detail::is_observable<Decayed>
                        {
                        };

                        template<class Observable, class DecayedObservable = rxu::decay_t<Observable>>
                        using observable_tag_t = typename DecayedObservable::observable_tag;

                        // extra indirection for vs2013 support
                        template<class Types, class = rxu::types_checked>
                        struct expand_observable_tags { struct type; };
                        template<class... ObservableN>
                        struct expand_observable_tags<rxu::types<ObservableN...>, rxu::types_checked_t<typename
ObservableN::observable_tag...>>
                        {
                                using type = rxu::types<typename ObservableN::observable_tag...>;
                        };
                        template<class... ObservableN>
                        using observable_tags_t = typename expand_observable_tags<rxu::types<ObservableN...>>::type;

                        template<class... ObservableN>
                        using all_observables = rxu::all_true_type<is_observable<ObservableN>...>;

                        struct tag_dynamic_connectable_observable : public tag_dynamic_observable {};
```

```
template<class T>
class is_dynamic_connectable_observable
{
	struct not_void {};
	template<class C>
	static typename C::dynamic_observable_tag* check(int);
	template<class C>
	static not_void check(...);
public:
	static const bool value = std::is_convertible<decltype(check<rxu::decay_t<T>>(0)),
tag_dynamic_connectable_observable*>::value;
};

template<class T>
class dynamic_connectable_observable;

template<class T,
class SourceObservable = typename std::conditional<std::is_same<T, void>::value,
	void, dynamic_connectable_observable<T >> ::type>
	class connectable_observable;

	struct tag_connectable_observable : public tag_observable {};
	template<class T>
	class is_connectable_observable
	{
		template<class C>
		static typename C::observable_tag check(int);
		template<class C>
		static void check(...);
	public:
		static const bool value = std::is_convertible<decltype(check<rxu::decay_t<T>>(0)),
tag_connectable_observable>::value;
	};

	struct tag_dynamic_grouped_observable : public tag_dynamic_observable {};

	template<class T>
	class is_dynamic_grouped_observable
	{
		struct not_void {};
		template<class C>
		static typename C::dynamic_observable_tag* check(int);
		template<class C>
		static not_void check(...);
	public:
		static const bool value = std::is_convertible<decltype(check<rxu::decay_t<T>>(0)),
tag_dynamic_grouped_observable*>::value;
	};

	template<class K, class T>
	class dynamic_grouped_observable;

	template<class K, class T,
	class SourceObservable = typename std::conditional<std::is_same<T, void>::value,
		void, dynamic_grouped_observable<K, T >> ::type>
		class grouped_observable;

		template<class K, class T, class Source>
		grouped_observable<K, T> make_dynamic_grouped_observable(Source&& s);

		struct tag_grouped_observable : public tag_observable {};
		template<class T>
		class is_grouped_observable
		{
			template<class C>
			static typename C::observable_tag check(int);
			template<class C>
			static void check(...);
		public:
			static const bool value =
std::is_convertible<decltype(check<rxu::decay_t<T>>(0)), tag_grouped_observable>::value;
		};

		template<class Source, class Function>
		struct is_operator_factory_for {
			using function_type = rxu::decay_t<Function>;
			using source_type = rxu::decay_t<Source>;

			// check methods instead of void_t for vs2013 support

			struct tag_not_valid;
```

```
                                                template<class CS, class CO>
                                                static auto check(int) -> decltype((*(CS*)nullptr)((*(CO*)nullptr)));
                                                template<class CS, class CO>
                                                static tag_not_valid check(...);

                                                using type = decltype(check<function_type, source_type>(0));

                                                static const bool value = !std::is_same<type, tag_not_valid>::value &&
is_observable<source_type>::value;
                                        };

                                        //
                                        // this type is the default used by operators that subscribe to
                                        // multiple sources. It assumes that the sources are already synchronized
                                        //
                                        struct identity_observable
                                        {
                                                template<class Observable>
                                                auto operator()(Observable o)
                                                        -> Observable {
                                                        return  std::move(o);
                                                        static_assert(is_observable<Observable>::value, "only support
observables");
                                                }
                                        };

                                        template<class T>
                                        struct identity_for
                                        {
                                                T operator()(T t) {
                                                        return  std::move(t);
                                                }
                                        };

}

#endif


#if !defined(RXCPP_RX_SUBSCRIPTION_HPP)
#define RXCPP_RX_SUBSCRIPTION_HPP

//_include "rx-includes.hpp"

namespace rxcpp {

        namespace detail {

                template<class F>
                struct is_unsubscribe_function
                {
                        struct not_void {};
                        template<class CF>
                        static auto check(int) -> decltype((*(CF*)nullptr)());
                        template<class CF>
                        static not_void check(...);

                        static const bool value = std::is_same<decltype(check<rxu::decay_t<F>>(0)), void>::value;
                };

        }

        struct tag_subscription {};
        struct subscription_base { typedef tag_subscription subscription_tag; };
        template<class T>
        class is_subscription
        {
                template<class C>
                static typename C::subscription_tag* check(int);
                template<class C>
                static void check(...);
        public:
                static const bool value = std::is_convertible<decltype(check<rxu::decay_t<T>>(0)), tag_subscription*>::value;
        };

        template<class Unsubscribe>
        class static_subscription
        {
                typedef rxu::decay_t<Unsubscribe> unsubscribe_call_type;
                unsubscribe_call_type unsubscribe_call;
                static_subscription()
                {
```

```
        }
public:
        static_subscription(const static_subscription& o)
                : unsubscribe_call(o.unsubscribe_call)
        {
        }
        static_subscription(static_subscription&& o)
                : unsubscribe_call(std::move(o.unsubscribe_call))
        {
        }
        static_subscription(unsubscribe_call_type s)
                : unsubscribe_call(std::move(s))
        {
        }
        void unsubscribe() const {
                unsubscribe_call();
        }
};

class subscription : public subscription_base
{
        class base_subscription_state : public std::enable_shared_from_this<base_subscription_state>
        {
                base_subscription_state();
        public:

                explicit base_subscription_state(bool initial)
                        : issubscribed(initial)
                {
                }
                virtual ~base_subscription_state() {}
                virtual void unsubscribe() {
                }
                std::atomic<bool> issubscribed;
        };
public:
        typedef std::weak_ptr<base_subscription_state> weak_state_type;

private:
        template<class I>
        struct subscription_state : public base_subscription_state
        {
                typedef rxu::decay_t<I> inner_t;
                subscription_state(inner_t i)
                        : base_subscription_state(true)
                        , inner(std::move(i))
                {
                }
                virtual void unsubscribe() {
                        if (issubscribed.exchange(false)) {
                                trace_activity().unsubscribe_enter(*this);
                                inner.unsubscribe();
                                trace_activity().unsubscribe_return(*this);
                        }
                }
                inner_t inner;
        };

protected:
        std::shared_ptr<base_subscription_state> state;

        friend bool operator<(const subscription&, const subscription&);
        friend bool operator==(const subscription&, const subscription&);

private:
        subscription(weak_state_type w)
                : state(w.lock())
        {
                if (!state) {
                        std::terminate();
                }
        }
public:

        subscription()
                : state(std::make_shared<base_subscription_state>(false))
        {
                if (!state) {
                        std::terminate();
                }
        }
        template<class U>
```

```
                explicit subscription(U u, typename std::enable_if<!is_subscription<U>::value, void**>::type = nullptr)
                        : state(std::make_shared<subscription_state<U>>(std::move(u)))
                {
                        if (!state) {
                                std::terminate();
                        }
                }
                template<class U>
                explicit subscription(U u, typename std::enable_if<!std::is_same<subscription, U>::value && is_subscription<U>::value,
void**>::type = nullptr)
                        // intentionally slice
                        : state(std::move((*static_cast<subscription*>(&u)).state))
                {
                        if (!state) {
                                std::terminate();
                        }
                }
                subscription(const subscription& o)
                        : state(o.state)
                {
                        if (!state) {
                                std::terminate();
                        }
                }
                subscription(subscription&& o)
                        : state(std::move(o.state))
                {
                        if (!state) {
                                std::terminate();
                        }
                }
                subscription& operator=(subscription o) {
                        state = std::move(o.state);
                        return *this;
                }
                bool is_subscribed() const {
                        if (!state) {
                                std::terminate();
                        }
                        return state->issubscribed;
                }
                void unsubscribe() const {
                        if (!state) {
                                std::terminate();
                        }
                        auto keepAlive = state;
                        state->unsubscribe();
                }

                weak_state_type get_weak() {
                        return state;
                }
                static subscription lock(weak_state_type w) {
                        return subscription(w);
                }
        };

        inline bool operator<(const subscription& lhs, const subscription& rhs) {
                return lhs.state < rhs.state;
        }
        inline bool operator==(const subscription& lhs, const subscription& rhs) {
                return lhs.state == rhs.state;
        }
        inline bool operator!=(const subscription& lhs, const subscription& rhs) {
                return !(lhs == rhs);
        }


        inline auto make_subscription()
                ->      subscription {
                return  subscription();
        }
        template<class I>
        auto make_subscription(I&& i)
                -> typename std::enable_if<!is_subscription<I>::value && !detail::is_unsubscribe_function<I>::value,
                subscription>::type {
                return  subscription(std::forward<I>(i));
        }
        template<class Unsubscribe>
        auto make_subscription(Unsubscribe&& u)
                -> typename std::enable_if<detail::is_unsubscribe_function<Unsubscribe>::value,
                subscription>::type {
```

```
        return  subscription(static_subscription<Unsubscribe>(std::forward<Unsubscribe>(u)));
    }

    class composite_subscription;

    namespace detail {

        struct tag_composite_subscription_empty {};

        class composite_subscription_inner
        {
        private:
            typedef subscription::weak_state_type weak_subscription;
            struct composite_subscription_state : public std::enable_shared_from_this<composite_subscription_state>
            {
                std::set<subscription> subscriptions;
                std::mutex lock;
                std::atomic<bool> issubscribed;

                ~composite_subscription_state()
                {
                    std::unique_lock<decltype(lock)> guard(lock);
                    subscriptions.clear();
                }

                composite_subscription_state()
                    : issubscribed(true)
                {
                }
                composite_subscription_state(tag_composite_subscription_empty)
                    : issubscribed(false)
                {
                }

                inline weak_subscription add(subscription s) {
                    if (!issubscribed) {
                        s.unsubscribe();
                    }
                    else if (s.is_subscribed()) {
                        std::unique_lock<decltype(lock)> guard(lock);
                        subscriptions.insert(s);
                    }
                    return s.get_weak();
                }

                inline void remove(weak_subscription w) {
                    if (issubscribed && !w.expired()) {
                        auto s = subscription::lock(w);
                        std::unique_lock<decltype(lock)> guard(lock);
                        subscriptions.erase(std::move(s));
                    }
                }

                inline void clear() {
                    if (issubscribed) {
                        std::unique_lock<decltype(lock)> guard(lock);

                        std::set<subscription> v(std::move(subscriptions));
                        guard.unlock();
                        std::for_each(v.begin(), v.end(),
                            [](const subscription& s) {
                            s.unsubscribe(); });
                    }
                }

                inline void unsubscribe() {
                    if (issubscribed.exchange(false)) {
                        std::unique_lock<decltype(lock)> guard(lock);

                        std::set<subscription> v(std::move(subscriptions));
                        guard.unlock();
                        std::for_each(v.begin(), v.end(),
                            [](const subscription& s) {
                            s.unsubscribe(); });
                    }
                }
            };

        public:
            typedef std::shared_ptr<composite_subscription_state> shared_state_type;

        protected:
```

```
                mutable shared_state_type state;

        public:
                composite_subscription_inner()
                        : state(std::make_shared<composite_subscription_state>())
                {
                }
                composite_subscription_inner(tag_composite_subscription_empty et)
                        : state(std::make_shared<composite_subscription_state>(et))
                {
                }

                composite_subscription_inner(const composite_subscription_inner& o)
                        : state(o.state)
                {
                        if (!state) {
                                std::terminate();
                        }
                }
                composite_subscription_inner(composite_subscription_inner&& o)
                        : state(std::move(o.state))
                {
                        if (!state) {
                                std::terminate();
                        }
                }

                composite_subscription_inner& operator=(composite_subscription_inner o)
                {
                        state = std::move(o.state);
                        if (!state) {
                                std::terminate();
                        }
                        return *this;
                }

                inline weak_subscription add(subscription s) const {
                        if (!state) {
                                std::terminate();
                        }
                        return state->add(std::move(s));
                }
                inline void remove(weak_subscription w) const {
                        if (!state) {
                                std::terminate();
                        }
                        state->remove(std::move(w));
                }
                inline void clear() const {
                        if (!state) {
                                std::terminate();
                        }
                        state->clear();
                }
                inline void unsubscribe() {
                        if (!state) {
                                std::terminate();
                        }
                        state->unsubscribe();
                }
        };

        inline composite_subscription shared_empty();

}

/*!
\brief controls lifetime for scheduler::schedule and observable<T, SourceOperator>::subscribe.

\ingroup group-core

*/
class composite_subscription
        : protected detail::composite_subscription_inner
        , public subscription
{
        typedef detail::composite_subscription_inner inner_type;
public:
        typedef subscription::weak_state_type weak_subscription;

        composite_subscription(detail::tag_composite_subscription_empty et)
                : inner_type(et)
```

```
                , subscription() // use empty base
        {
        }

public:

        composite_subscription()
                : inner_type()
                , subscription(*static_cast<const inner_type* const>(this))
        {
        }

        composite_subscription(const composite_subscription& o)
                : inner_type(o)
                , subscription(static_cast<const subscription&>(o))
        {
        }
        composite_subscription(composite_subscription&& o)
                : inner_type(std::move(o))
                , subscription(std::move(static_cast<subscription&>(o)))
        {
        }

        composite_subscription& operator=(composite_subscription o)
        {
                inner_type::operator=(std::move(o));
                subscription::operator=(std::move(*static_cast<subscription*>(&o)));
                return *this;
        }

        static inline composite_subscription empty() {
                return detail::shared_empty();
        }

        using subscription::is_subscribed;
        using subscription::unsubscribe;

        using inner_type::clear;

        inline weak_subscription add(subscription s) const {
                if (s == static_cast<const subscription&>(*this)) {
                        // do not nest the same subscription
                        std::terminate();
                        //return s.get_weak();
                }
                auto that = this->subscription::state.get();
                trace_activity().subscription_add_enter(*that, s);
                auto w = inner_type::add(std::move(s));
                trace_activity().subscription_add_return(*that);
                return w;
        }

        template<class F>
        auto add(F f) const
                -> typename std::enable_if<detail::is_unsubscribe_function<F>::value, weak_subscription>::type {
                return add(make_subscription(std::move(f)));
        }

        inline void remove(weak_subscription w) const {
                auto that = this->subscription::state.get();
                trace_activity().subscription_remove_enter(*that, w);
                inner_type::remove(w);
                trace_activity().subscription_remove_return(*that);
        }
};

inline bool operator<(const composite_subscription& lhs, const composite_subscription& rhs) {
        return static_cast<const subscription&>(lhs) < static_cast<const subscription&>(rhs);
}
inline bool operator==(const composite_subscription& lhs, const composite_subscription& rhs) {
        return static_cast<const subscription&>(lhs) == static_cast<const subscription&>(rhs);
}
inline bool operator!=(const composite_subscription& lhs, const composite_subscription& rhs) {
        return !(lhs == rhs);
}

namespace detail {

        inline composite_subscription shared_empty() {
                static composite_subscription shared_empty = composite_subscription(tag_composite_subscription_empty());
                return shared_empty;
        }
```

```
	}

	template<class T>
	class resource : public subscription_base
	{
	public:
		typedef typename composite_subscription::weak_subscription weak_subscription;

		resource()
			: lifetime(composite_subscription())
			, value(std::make_shared<rxu::detail::maybe<T>>())
		{
		}

		explicit resource(T t, composite_subscription cs = composite_subscription())
			: lifetime(std::move(cs))
			, value(std::make_shared<rxu::detail::maybe<T>>(rxu::detail::maybe<T>(std::move(t))))
		{
			auto localValue = value;
			lifetime.add(
				[localValue](){
				localValue->reset();
			}
			);
		}

		T& get() {
			return value.get()->get();
		}
		composite_subscription& get_subscription() {
			return lifetime;
		}

		bool is_subscribed() const {
			return lifetime.is_subscribed();
		}
		weak_subscription add(subscription s) const {
			return lifetime.add(std::move(s));
		}
		template<class F>
		auto add(F f) const
			-> typename std::enable_if<detail::is_unsubscribe_function<F>::value, weak_subscription>::type {
			return lifetime.add(make_subscription(std::move(f)));
		}
		void remove(weak_subscription w) const {
			return lifetime.remove(std::move(w));
		}
		void clear() const {
			return lifetime.clear();
		}
		void unsubscribe() const {
			return lifetime.unsubscribe();
		}

	protected:
		composite_subscription lifetime;
		std::shared_ptr<rxu::detail::maybe<T>> value;
	};

}

#endif


#if !defined(RXCPP_RX_OBSERVER_HPP)
#define RXCPP_RX_OBSERVER_HPP

//_include "rx-includes.hpp"

namespace rxcpp {


	template<class T>
	struct observer_base
	{
		typedef T value_type;
		typedef tag_observer observer_tag;
	};

	namespace detail {
		template<class T>
```

```
struct OnNextEmpty
{
	void operator()(const T&) const {}
};
struct OnErrorEmpty
{
	void operator()(std::exception_ptr) const {
		// error implicitly ignored, abort
		std::terminate();
	}
};
struct OnErrorIgnore
{
	void operator()(std::exception_ptr) const {
	}
};
struct OnCompletedEmpty
{
	void operator()() const {}
};

template<class T, class State, class OnNext>
struct OnNextForward
{
	using state_t = rxu::decay_t<State>;
	using onnext_t = rxu::decay_t<OnNext>;
	OnNextForward() : onnext() {}
	explicit OnNextForward(onnext_t on) : onnext(std::move(on)) {}
	onnext_t onnext;
	void operator()(state_t& s, T& t) const {
		onnext(s, t);
	}
	void operator()(state_t& s, T&& t) const {
		onnext(s, t);
	}
};
template<class T, class State>
struct OnNextForward<T, State, void>
{
	using state_t = rxu::decay_t<State>;
	OnNextForward() {}
	void operator()(state_t& s, T& t) const {
		s.on_next(t);
	}
	void operator()(state_t& s, T&& t) const {
		s.on_next(t);
	}
};

template<class State, class OnError>
struct OnErrorForward
{
	using state_t = rxu::decay_t<State>;
	using onerror_t = rxu::decay_t<OnError>;
	OnErrorForward() : onerror() {}
	explicit OnErrorForward(onerror_t oe) : onerror(std::move(oe)) {}
	onerror_t onerror;
	void operator()(state_t& s, std::exception_ptr ep) const {
		onerror(s, ep);
	}
};
template<class State>
struct OnErrorForward<State, void>
{
	using state_t = rxu::decay_t<State>;
	OnErrorForward() {}
	void operator()(state_t& s, std::exception_ptr ep) const {
		s.on_error(ep);
	}
};

template<class State, class OnCompleted>
struct OnCompletedForward
{
	using state_t = rxu::decay_t<State>;
	using oncompleted_t = rxu::decay_t<OnCompleted>;
	OnCompletedForward() : oncompleted() {}
	explicit OnCompletedForward(oncompleted_t oc) : oncompleted(std::move(oc)) {}
	oncompleted_t oncompleted;
	void operator()(state_t& s) const {
		oncompleted(s);
	}
```

```
        };
        template<class State>
        struct OnCompletedForward<State, void>
        {
                OnCompletedForward() {}
                void operator()(State& s) const {
                        s.on_completed();
                }
        };

        template<class T, class F>
        struct is_on_next_of
        {
                struct not_void {};
                template<class CT, class CF>
                static auto check(int) -> decltype((*(CF*)nullptr)(*(CT*)nullptr));
                template<class CT, class CF>
                static not_void check(...);

                typedef decltype(check<T, rxu::decay_t<F>>(0)) detail_result;
                static const bool value = std::is_same<detail_result, void>::value;
        };

        template<class F>
        struct is_on_error
        {
                struct not_void {};
                template<class CF>
                static auto check(int) -> decltype((*(CF*)nullptr)(*(std::exception_ptr*)nullptr));
                template<class CF>
                static not_void check(...);

                static const bool value = std::is_same<decltype(check<rxu::decay_t<F>>(0)), void>::value;
        };

        template<class State, class F>
        struct is_on_error_for
        {
                struct not_void {};
                template<class CF>
                static auto check(int) -> decltype((*(CF*)nullptr)(*(State*)nullptr, *(std::exception_ptr*)nullptr));
                template<class CF>
                static not_void check(...);

                static const bool value = std::is_same<decltype(check<rxu::decay_t<F>>(0)), void>::value;
        };

        template<class F>
        struct is_on_completed
        {
                struct not_void {};
                template<class CF>
                static auto check(int) -> decltype((*(CF*)nullptr)());
                template<class CF>
                static not_void check(...);

                static const bool value = std::is_same<decltype(check<rxu::decay_t<F>>(0)), void>::value;
        };

    }


    /*!
    \brief consumes values from an observable using `State` that may implement on_next, on_error and on_completed with optional overrides of each function.

    \tparam T           - the type of value in the stream
    \tparam State       - the type of the stored state
    \tparam OnNext      - the type of a function that matches `void(State&, T)`. Called 0 or more times. If `void` State::on_next will be called.
    \tparam OnError     - the type of a function that matches `void(State&, std::exception_ptr)`. Called 0 or 1 times, no further calls will be made. If `void` State::on_error will be called.
    \tparam OnCompleted - the type of a function that matches `void(State&)`. Called 0 or 1 times, no further calls will be made. If `void` State::on_completed will be called.

    \ingroup group-core

    */
    template<class T, class State, class OnNext, class OnError, class OnCompleted>
    class observer : public observer_base<T>
    {
    public:
```

```
        using this_type = observer<T, State, OnNext, OnError, OnCompleted>;
        using state_t = rxu::decay_t<State>;
        using on_next_t = typename std::conditional<
            !std::is_same<void, OnNext>::value,
            rxu::decay_t<OnNext>,
            detail::OnNextForward<T, State, OnNext >> ::type;
        using on_error_t = typename std::conditional<
            !std::is_same<void, OnError>::value,
            rxu::decay_t<OnError>,
            detail::OnErrorForward<State, OnError >> ::type;
        using on_completed_t = typename std::conditional<
            !std::is_same<void, OnCompleted>::value,
            rxu::decay_t<OnCompleted>,
            detail::OnCompletedForward<State, OnCompleted >> ::type;

    private:
        mutable state_t state;
        on_next_t onnext;
        on_error_t onerror;
        on_completed_t oncompleted;

    public:

        explicit observer(state_t s, on_next_t n = on_next_t(), on_error_t e = on_error_t(), on_completed_t c = on_completed_t())
            : state(std::move(s))
            , onnext(std::move(n))
            , onerror(std::move(e))
            , oncompleted(std::move(c))
        {
        }
        explicit observer(state_t s, on_next_t n, on_completed_t c)
            : state(std::move(s))
            , onnext(std::move(n))
            , onerror(on_error_t())
            , oncompleted(std::move(c))
        {
        }
        observer(const this_type& o)
            : state(o.state)
            , onnext(o.onnext)
            , onerror(o.onerror)
            , oncompleted(o.oncompleted)
        {
        }
        observer(this_type&& o)
            : state(std::move(o.state))
            , onnext(std::move(o.onnext))
            , onerror(std::move(o.onerror))
            , oncompleted(std::move(o.oncompleted))
        {
        }
        this_type& operator=(this_type o) {
            state = std::move(o.state);
            onnext = std::move(o.onnext);
            onerror = std::move(o.onerror);
            oncompleted = std::move(o.oncompleted);
            return *this;
        }

        void on_next(T& t) const {
            onnext(state, t);
        }
        void on_next(T&& t) const {
            onnext(state, std::move(t));
        }
        void on_error(std::exception_ptr e) const {
            onerror(state, e);
        }
        void on_completed() const {
            oncompleted(state);
        }
        observer<T> as_dynamic() const {
            return observer<T>(*this);
        }
    };

    /*!
    \brief consumes values from an observable using default empty method implementations with optional overrides of each function.

    \tparam T           - the type of value in the stream
    \tparam OnNext      - the type of a function that matches `void(T)`. Called 0 or more times. If `void` OnNextEmpty<T> is used.
```

```
        \tparam OnError      - the type of a function that matches `void(std::exception_ptr)`. Called 0 or 1 times, no further calls will be made. If `void` OnErrorEmpty is used.
        \tparam OnCompleted  - the type of a function that matches `void()`. Called 0 or 1 times, no further calls will be made. If `void` OnCompletedEmpty is used.

        \ingroup group-core

        */
        template<class T, class OnNext, class OnError, class OnCompleted>
        class observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted> : public observer_base<T>
        {
        public:
                using this_type = observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>;
                using on_next_t = typename std::conditional<
                        !std::is_same<void, OnNext>::value,
                        rxu::decay_t<OnNext>,
                        detail::OnNextEmpty<T >> ::type;
                using on_error_t = typename std::conditional<
                        !std::is_same<void, OnError>::value,
                        rxu::decay_t<OnError>,
                        detail::OnErrorEmpty>::type;
                using on_completed_t = typename std::conditional<
                        !std::is_same<void, OnCompleted>::value,
                        rxu::decay_t<OnCompleted>,
                        detail::OnCompletedEmpty>::type;

        private:
                on_next_t onnext;
                on_error_t onerror;
                on_completed_t oncompleted;

        public:
                static_assert(detail::is_on_next_of<T, on_next_t>::value, "Function supplied for on_next must be a function with the signature void(T);");
                static_assert(detail::is_on_error<on_error_t>::value, "Function supplied for on_error must be a function with the signature void(std::exception_ptr);");
                static_assert(detail::is_on_completed<on_completed_t>::value, "Function supplied for on_completed must be a function with the signature void();");

                observer()
                        : onnext(on_next_t())
                        , onerror(on_error_t())
                        , oncompleted(on_completed_t())
                {
                }

                explicit observer(on_next_t n, on_error_t e = on_error_t(), on_completed_t c = on_completed_t())
                        : onnext(std::move(n))
                        , onerror(std::move(e))
                        , oncompleted(std::move(c))
                {
                }
                observer(const this_type& o)
                        : onnext(o.onnext)
                        , onerror(o.onerror)
                        , oncompleted(o.oncompleted)
                {
                }
                observer(this_type&& o)
                        : onnext(std::move(o.onnext))
                        , onerror(std::move(o.onerror))
                        , oncompleted(std::move(o.oncompleted))
                {
                }
                this_type& operator=(this_type o) {
                        onnext = std::move(o.onnext);
                        onerror = std::move(o.onerror);
                        oncompleted = std::move(o.oncompleted);
                        return *this;
                }

                void on_next(T& t) const {
                        onnext(t);
                }
                void on_next(T&& t) const {
                        onnext(std::move(t));
                }
                void on_error(std::exception_ptr e) const {
                        onerror(e);
                }
                void on_completed() const {
                        oncompleted();
```

```
        }
        observer<T> as_dynamic() const {
                return observer<T>(*this);
        }
};

namespace detail
{

        template<class T>
        struct virtual_observer : public std::enable_shared_from_this<virtual_observer<T>>
        {
                virtual ~virtual_observer() {}
                virtual void on_next(T&) const {};
                virtual void on_next(T&&) const {};
                virtual void on_error(std::exception_ptr) const {};
                virtual void on_completed() const {};
        };

        template<class T, class Observer>
        struct specific_observer : public virtual_observer<T>
        {
                explicit specific_observer(Observer o)
                : destination(std::move(o))
                {
                }

                Observer destination;
                virtual void on_next(T& t) const {
                        destination.on_next(t);
                }
                virtual void on_next(T&& t) const {
                        destination.on_next(std::move(t));
                }
                virtual void on_error(std::exception_ptr e) const {
                        destination.on_error(e);
                }
                virtual void on_completed() const {
                        destination.on_completed();
                }
        };

}

/*!
\brief consumes values from an observable using type-forgetting (shared allocated state with virtual methods)

\tparam T           - the type of value in the stream

\ingroup group-core

*/
template<class T>
class observer<T, void, void, void, void> : public observer_base<T>
{
public:
        typedef tag_dynamic_observer dynamic_observer_tag;

private:
        using this_type = observer<T, void, void, void, void>;
        using base_type = observer_base<T>;
        using virtual_observer = detail::virtual_observer<T>;

        std::shared_ptr<virtual_observer> destination;

        template<class Observer>
        static auto make_destination(Observer o)
                -> std::shared_ptr<virtual_observer> {
                return std::make_shared<detail::specific_observer<T, Observer>>(std::move(o));
        }

public:
        observer()
        {
        }
        observer(const this_type& o)
                : destination(o.destination)
        {
        }
        observer(this_type&& o)
                : destination(std::move(o.destination))
        {
```

```
		}

		template<class Observer>
		explicit observer(Observer o)
			: destination(make_destination(std::move(o)))
		{
		}

		this_type& operator=(this_type o) {
			destination = std::move(o.destination);
			return *this;
		}

		// perfect forwarding delays the copy of the value.
		template<class V>
		void on_next(V&& v) const {
			if (destination) {
				destination->on_next(std::forward<V>(v));
			}
		}
		void on_error(std::exception_ptr e) const {
			if (destination) {
				destination->on_error(e);
			}
		}
		void on_completed() const {
			if (destination) {
				destination->on_completed();
			}
		}

		observer<T> as_dynamic() const {
			return *this;
		}
	};

	template<class T, class DefaultOnError = detail::OnErrorEmpty>
	auto make_observer()
		->	observer<T, detail::stateless_observer_tag, detail::OnNextEmpty<T>, DefaultOnError> {
		return	observer<T, detail::stateless_observer_tag, detail::OnNextEmpty<T>, DefaultOnError>();
	}

	template<class T, class DefaultOnError = detail::OnErrorEmpty, class U, class State, class OnNext, class OnError, class
OnCompleted>
	auto make_observer(observer<U, State, OnNext, OnError, OnCompleted> o)
		->	observer<T, State, OnNext, OnError, OnCompleted> {
		return	observer<T, State, OnNext, OnError, OnCompleted>(std::move(o));
	}
	template<class T, class DefaultOnError = detail::OnErrorEmpty, class Observer>
	auto make_observer(Observer ob)
		-> typename std::enable_if<
		!detail::is_on_next_of<T, Observer>::value &&
		!detail::is_on_error<Observer>::value &&
		is_observer<Observer>::value,
		Observer>::type {
		return	std::move(ob);
	}
	template<class T, class DefaultOnError = detail::OnErrorEmpty, class Observer>
	auto make_observer(Observer ob)
		-> typename std::enable_if<
		!detail::is_on_next_of<T, Observer>::value &&
		!detail::is_on_error<Observer>::value &&
		!is_observer<Observer>::value,
		observer<T, Observer >> ::type {
		return	observer<T, Observer>(std::move(ob));
	}
	template<class T, class DefaultOnError = detail::OnErrorEmpty, class OnNext>
	auto make_observer(OnNext on)
		-> typename std::enable_if<
		detail::is_on_next_of<T, OnNext>::value,
		observer<T, detail::stateless_observer_tag, OnNext, DefaultOnError >> ::type {
		return	observer<T, detail::stateless_observer_tag, OnNext, DefaultOnError>(
			std::move(on));
	}
	template<class T, class DefaultOnError = detail::OnErrorEmpty, class OnError>
	auto make_observer(OnError oe)
		-> typename std::enable_if<
		detail::is_on_error<OnError>::value,
		observer<T, detail::stateless_observer_tag, detail::OnNextEmpty<T>, OnError >> ::type {
		return	observer<T, detail::stateless_observer_tag, detail::OnNextEmpty<T>, OnError>(
			detail::OnNextEmpty<T>(), std::move(oe));
	}
```

```
template<class T, class DefaultOnError = detail::OnErrorEmpty, class OnNext, class OnError>
auto make_observer(OnNext on, OnError oe)
	-> typename std::enable_if<
	detail::is_on_next_of<T, OnNext>::value &&
	detail::is_on_error<OnError>::value,
	observer<T, detail::stateless_observer_tag, OnNext, OnError >> ::type {
	return  observer<T, detail::stateless_observer_tag, OnNext, OnError>(
			std::move(on), std::move(oe));
}
template<class T, class DefaultOnError = detail::OnErrorEmpty, class OnNext, class OnCompleted>
auto make_observer(OnNext on, OnCompleted oc)
	-> typename std::enable_if<
	detail::is_on_next_of<T, OnNext>::value &&
	detail::is_on_completed<OnCompleted>::value,
	observer<T, detail::stateless_observer_tag, OnNext, DefaultOnError, OnCompleted >> ::type {
	return  observer<T, detail::stateless_observer_tag, OnNext, DefaultOnError, OnCompleted>(
			std::move(on), DefaultOnError(), std::move(oc));
}
template<class T, class DefaultOnError = detail::OnErrorEmpty, class OnNext, class OnError, class OnCompleted>
auto make_observer(OnNext on, OnError oe, OnCompleted oc)
	-> typename std::enable_if<
	detail::is_on_next_of<T, OnNext>::value &&
	detail::is_on_error<OnError>::value &&
	detail::is_on_completed<OnCompleted>::value,
	observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted >> ::type {
	return  observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>(
			std::move(on), std::move(oe), std::move(oc));
}


template<class T, class State, class OnNext>
auto make_observer(State os, OnNext on)
	-> typename std::enable_if<
	!detail::is_on_next_of<T, State>::value &&
	!detail::is_on_error<State>::value,
	observer<T, State, OnNext >> ::type {
	return  observer<T, State, OnNext>(
			std::move(os), std::move(on));
}
template<class T, class State, class OnError>
auto make_observer(State os, OnError oe)
	-> typename std::enable_if<
	!detail::is_on_next_of<T, State>::value &&
	!detail::is_on_error<State>::value &&
	detail::is_on_error_for<State, OnError>::value,
	observer<T, State, detail::OnNextEmpty<T>, OnError >> ::type {
	return  observer<T, State, detail::OnNextEmpty<T>, OnError>(
			std::move(os), detail::OnNextEmpty<T>(), std::move(oe));
}
template<class T, class State, class OnNext, class OnError>
auto make_observer(State os, OnNext on, OnError oe)
	-> typename std::enable_if<
	!detail::is_on_next_of<T, State>::value &&
	!detail::is_on_error<State>::value &&
	detail::is_on_error_for<State, OnError>::value,
	observer<T, State, OnNext, OnError >> ::type {
	return  observer<T, State, OnNext, OnError>(
			std::move(os), std::move(on), std::move(oe));
}
template<class T, class State, class OnNext, class OnCompleted>
auto make_observer(State os, OnNext on, OnCompleted oc)
	-> typename std::enable_if<
	!detail::is_on_next_of<T, State>::value &&
	!detail::is_on_error<State>::value,
	observer<T, State, OnNext, void, OnCompleted >> ::type {
	return  observer<T, State, OnNext, void, OnCompleted>(
			std::move(os), std::move(on), std::move(oc));
}
template<class T, class State, class OnNext, class OnError, class OnCompleted>
auto make_observer(State os, OnNext on, OnError oe, OnCompleted oc)
	-> typename std::enable_if<
	!detail::is_on_next_of<T, State>::value &&
	!detail::is_on_error<State>::value &&
	detail::is_on_error_for<State, OnError>::value,
	observer<T, State, OnNext, OnError, OnCompleted >> ::type {
	return  observer<T, State, OnNext, OnError, OnCompleted>(
			std::move(os), std::move(on), std::move(oe), std::move(oc));
}

template<class T, class Observer>
auto make_observer_dynamic(Observer o)
	-> typename std::enable_if<
```

```
                !detail::is_on_next_of<T, Observer>::value,
                observer<T >> ::type {
                return  observer<T>(std::move(o));
        }
        template<class T, class OnNext>
        auto make_observer_dynamic(OnNext&& on)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value,
                observer<T >> ::type {
                return  observer<T>(
                        make_observer<T>(std::forward<OnNext>(on)));
        }
        template<class T, class OnNext, class OnError>
        auto make_observer_dynamic(OnNext&& on, OnError&& oe)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value,
                observer<T >> ::type {
                return  observer<T>(
                        make_observer<T>(std::forward<OnNext>(on), std::forward<OnError>(oe)));
        }
        template<class T, class OnNext, class OnCompleted>
        auto make_observer_dynamic(OnNext&& on, OnCompleted&& oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_completed<OnCompleted>::value,
                observer<T >> ::type {
                return  observer<T>(
                        make_observer<T>(std::forward<OnNext>(on), std::forward<OnCompleted>(oc)));
        }
        template<class T, class OnNext, class OnError, class OnCompleted>
        auto make_observer_dynamic(OnNext&& on, OnError&& oe, OnCompleted&& oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value &&
                detail::is_on_completed<OnCompleted>::value,
                observer<T >> ::type {
                return  observer<T>(
                        make_observer<T>(std::forward<OnNext>(on), std::forward<OnError>(oe), std::forward<OnCompleted>(oc)));
        }

        namespace detail {

                template<class F>
                struct maybe_from_result
                {
                        typedef decltype((*(F*)nullptr)()) decl_result_type;
                        typedef rxu::decay_t<decl_result_type> result_type;
                        typedef rxu::maybe<result_type> type;
                };

        }

        template<class F, class OnError>
        auto on_exception(const F& f, const OnError& c)
                ->  typename std::enable_if<detail::is_on_error<OnError>::value, typename detail::maybe_from_result<F>::type>::type {
                typename detail::maybe_from_result<F>::type r;
                try {
                        r.reset(f());
                }
                catch (...) {
                        c(std::current_exception());
                }
                return r;
        }

        template<class F, class Subscriber>
        auto on_exception(const F& f, const Subscriber& s)
                ->  typename std::enable_if<is_subscriber<Subscriber>::value, typename detail::maybe_from_result<F>::type>::type {
                typename detail::maybe_from_result<F>::type r;
                try {
                        r.reset(f());
                }
                catch (...) {
                        s.on_error(std::current_exception());
                }
                return r;
        }

}

#endif
```

```
#if !defined(RXCPP_RX_SCHEDULER_HPP)
#define RXCPP_RX_SCHEDULER_HPP

// include "rx-includes.hpp"

namespace rxcpp {

	namespace schedulers {

		class worker_interface;
		class scheduler_interface;

		namespace detail {

			class action_type;
			typedef std::shared_ptr<action_type> action_ptr;

			typedef std::shared_ptr<worker_interface> worker_interface_ptr;
			typedef std::shared_ptr<const worker_interface> const_worker_interface_ptr;

			typedef std::weak_ptr<worker_interface> worker_interface_weak_ptr;
			typedef std::weak_ptr<const worker_interface> const_worker_interface_weak_ptr;

			typedef std::shared_ptr<scheduler_interface> scheduler_interface_ptr;
			typedef std::shared_ptr<const scheduler_interface> const_scheduler_interface_ptr;

			inline action_ptr shared_empty() {
				static action_ptr shared_empty = std::make_shared<detail::action_type>();
				return shared_empty;
			}

		}

		// It is essential to keep virtual function calls out of an inner loop.
		// To make tail-recursion work efficiently the recursion objects create
		// a space on the stack inside the virtual function call in the actor that
		// allows the callback and the scheduler to share stack space that records
		// the request and the allowance without any virtual calls in the loop.

		/// recursed is set on a schedulable by the action to allow the called
		/// function to request to be rescheduled.
		class recursed
		{
			bool& isrequested;
			recursed operator=(const recursed&);
		public:
			explicit recursed(bool& r)
				: isrequested(r)
			{
			}
			/// request to be rescheduled
			inline void operator()() const {
				isrequested = true;
			}
		};

		/// recurse is passed to the action by the scheduler.
		/// the action uses recurse to coordinate the scheduler and the function.
		class recurse
		{
			bool& isallowed;
			mutable bool isrequested;
			recursed requestor;
			recurse operator=(const recurse&);
		public:
			explicit recurse(bool& a)
				: isallowed(a)
				, isrequested(true)
				, requestor(isrequested)
			{
			}
			/// does the scheduler allow tail-recursion now?
			inline bool is_allowed() const {
				return isallowed;
			}
			/// did the function request to be recursed?
			inline bool is_requested() const {
				return isrequested;
			}
			/// reset the function request. call before each call to the function.
```

```
		inline void reset() const {
			isrequested = false;
		}
		/// get the recursed to set into the schedulable for the function to use to request recursion
		inline const recursed& get_recursed() const {
			return requestor;
		}
};

/// recursion is used by the scheduler to signal to each action whether tail recursion is allowed.
class recursion
{
		mutable bool isallowed;
		recurse recursor;
		recursion operator=(const recursion&);
public:
		recursion()
			: isallowed(true)
			, recursor(isallowed)
		{
		}
		explicit recursion(bool b)
			: isallowed(b)
			, recursor(isallowed)
		{
		}
		/// set whether tail-recursion is allowed
		inline void reset(bool b = true) const {
			isallowed = b;
		}
		/// get the recurse to pass into each action being called
		inline const recurse& get_recurse() const {
			return recursor;
		}
};


struct action_base
{
		typedef tag_action action_tag;
};

class schedulable;

/// action provides type-forgetting for a potentially recursive set of calls to a function that takes a schedulable
class action : public action_base
{
		typedef action this_type;
		detail::action_ptr inner;
public:
		action()
		{
		}
		explicit action(detail::action_ptr i)
			: inner(std::move(i))
		{
		}

		/// return the empty action
		inline static action empty() {
			return action(detail::shared_empty());
		}

		/// call the function
		inline void operator()(const schedulable& s, const recurse& r) const;
};

struct scheduler_base
{
		typedef std::chrono::steady_clock clock_type;
		typedef tag_scheduler scheduler_tag;
};

struct worker_base : public subscription_base
{
		typedef tag_worker worker_tag;
};

class worker_interface
		: public std::enable_shared_from_this<worker_interface>
{
		typedef worker_interface this_type;
```

```
public:
	typedef scheduler_base::clock_type clock_type;

	virtual ~worker_interface() {}

	virtual clock_type::time_point now() const = 0;

	virtual void schedule(const schedulable& scbl) const = 0;
	virtual void schedule(clock_type::time_point when, const schedulable& scbl) const = 0;
};

namespace detail {

	template<class F>
	struct is_action_function
	{
		struct not_void {};
		template<class CF>
		static auto check(int) -> decltype((*(CF*)nullptr)(*(schedulable*)nullptr));
		template<class CF>
		static not_void check(...);

		static const bool value = std::is_same<decltype(check<rxu::decay_t<F>>(0)), void>::value;
	};

}

class weak_worker;

/// a worker ensures that all scheduled actions on the same instance are executed in-order with no overlap
/// a worker ensures that all scheduled actions are unsubscribed when it is unsubscribed
/// some inner implementations will impose additional constraints on the execution of items.
class worker : public worker_base
{
	typedef worker this_type;
	detail::worker_interface_ptr inner;
	composite_subscription lifetime;
	friend bool operator==(const worker&, const worker&);
	friend class weak_worker;
public:
	typedef scheduler_base::clock_type clock_type;
	typedef composite_subscription::weak_subscription weak_subscription;

	worker()
	{
	}
	worker(composite_subscription cs, detail::const_worker_interface_ptr i)
		: inner(std::const_pointer_cast<worker_interface>(i))
		, lifetime(std::move(cs))
	{
	}
	worker(composite_subscription cs, worker o)
		: inner(o.inner)
		, lifetime(std::move(cs))
	{
	}

	inline const composite_subscription& get_subscription() const {
		return lifetime;
	}
	inline composite_subscription& get_subscription() {
		return lifetime;
	}

	// composite_subscription
	//
	inline bool is_subscribed() const {
		return lifetime.is_subscribed();
	}
	inline weak_subscription add(subscription s) const {
		return lifetime.add(std::move(s));
	}
	inline void remove(weak_subscription w) const {
		return lifetime.remove(std::move(w));
	}
	inline void clear() const {
		return lifetime.clear();
	}
	inline void unsubscribe() const {
		return lifetime.unsubscribe();
	}
```

```
                        // worker_interface
                        //
                        /// return the current time for this worker
                        inline clock_type::time_point now() const {
                                return inner->now();
                        }

                        /// insert the supplied schedulable to be run as soon as possible
                        inline void schedule(const schedulable& scbl) const {
                                // force rebinding scbl to this worker
                                schedule_rebind(scbl);
                        }

                        /// insert the supplied schedulable to be run at the time specified
                        inline void schedule(clock_type::time_point when, const schedulable& scbl) const {
                                // force rebinding scbl to this worker
                                schedule_rebind(when, scbl);
                        }

                        // helpers
                        //

                        /// insert the supplied schedulable to be run at now() + the delay specified
                        inline void schedule(clock_type::duration when, const schedulable& scbl) const {
                                // force rebinding scbl to this worker
                                schedule_rebind(now() + when, scbl);
                        }

                        /// insert the supplied schedulable to be run at the initial time specified and then again at initial + (N * period)
                        /// this will continue until the worker or schedulable is unsubscribed.
                        inline void schedule_periodically(clock_type::time_point initial, clock_type::duration period, const schedulable&
scbl) const {
                                // force rebinding scbl to this worker
                                schedule_periodically_rebind(initial, period, scbl);
                        }

                        /// insert the supplied schedulable to be run at now() + the initial delay specified and then again at now() + initial
+ (N * period)
                        /// this will continue until the worker or schedulable is unsubscribed.
                        inline void schedule_periodically(clock_type::duration initial, clock_type::duration period, const schedulable&
scbl) const {
                                // force rebinding scbl to this worker
                                schedule_periodically_rebind(now() + initial, period, scbl);
                        }

                        /// use the supplied arguments to make a schedulable and then insert it to be run
                        template<class Arg0, class... ArgN>
                        auto schedule(Arg0&& a0, ArgN&&... an) const
                                -> typename std::enable_if<
                                (detail::is_action_function<Arg0>::value ||
                                is_subscription<Arg0>::value) &&
                                !is_schedulable<Arg0>::value>::type;
                        template<class... ArgN>
                        /// use the supplied arguments to make a schedulable and then insert it to be run
                        void schedule_rebind(const schedulable& scbl, ArgN&&... an) const;

                        /// use the supplied arguments to make a schedulable and then insert it to be run
                        template<class Arg0, class... ArgN>
                        auto schedule(clock_type::time_point when, Arg0&& a0, ArgN&&... an) const
                                -> typename std::enable_if<
                                (detail::is_action_function<Arg0>::value ||
                                is_subscription<Arg0>::value) &&
                                !is_schedulable<Arg0>::value>::type;
                        /// use the supplied arguments to make a schedulable and then insert it to be run
                        template<class... ArgN>
                        void schedule_rebind(clock_type::time_point when, const schedulable& scbl, ArgN&&... an) const;

                        /// use the supplied arguments to make a schedulable and then insert it to be run
                        template<class Arg0, class... ArgN>
                        auto schedule_periodically(clock_type::time_point initial, clock_type::duration period, Arg0&& a0, ArgN&&...
an) const
                                -> typename std::enable_if<
                                (detail::is_action_function<Arg0>::value ||
                                is_subscription<Arg0>::value) &&
                                !is_schedulable<Arg0>::value>::type;
                        /// use the supplied arguments to make a schedulable and then insert it to be run
                        template<class... ArgN>
                        void schedule_periodically_rebind(clock_type::time_point initial, clock_type::duration period, const
schedulable& scbl, ArgN&&... an) const;
                };
```

```
inline bool operator==(const worker& lhs, const worker& rhs) {
	return lhs.inner == rhs.inner && lhs.lifetime == rhs.lifetime;
}
inline bool operator!=(const worker& lhs, const worker& rhs) {
	return !(lhs == rhs);
}

class weak_worker
{
	detail::worker_interface_weak_ptr inner;
	composite_subscription lifetime;

public:
	weak_worker()
	{
	}
	explicit weak_worker(worker& owner)
		: inner(owner.inner)
		, lifetime(owner.lifetime)
	{
	}

	worker lock() const {
		return worker(lifetime, inner.lock());
	}
};

class scheduler_interface
	: public std::enable_shared_from_this<scheduler_interface>
{
	typedef scheduler_interface this_type;

public:
	typedef scheduler_base::clock_type clock_type;

	virtual ~scheduler_interface() {}

	virtual clock_type::time_point now() const = 0;

	virtual worker create_worker(composite_subscription cs) const = 0;
};


struct schedulable_base :
	// public subscription_base, <- already in worker base
	public worker_base,
	public action_base
{
	typedef tag_schedulable schedulable_tag;
};

/*!
\brief allows functions to be called at specified times and possibly in other contexts.

\ingroup group-core

*/
class scheduler : public scheduler_base
{
	typedef scheduler this_type;
	detail::scheduler_interface_ptr inner;
	friend bool operator==(const scheduler&, const scheduler&);
public:
	typedef scheduler_base::clock_type clock_type;

	scheduler()
	{
	}
	explicit scheduler(detail::scheduler_interface_ptr i)
		: inner(std::move(i))
	{
	}
	explicit scheduler(detail::const_scheduler_interface_ptr i)
		: inner(std::const_pointer_cast<scheduler_interface>(i))
	{
	}

	/// return the current time for this scheduler
	inline clock_type::time_point now() const {
		return inner->now();
	}
	/// create a worker with a lifetime.
```

```
                /// when the worker is unsubscribed all scheduled items will be unsubscribed.
                /// items scheduled to a worker will be run one at a time.
                /// scheduling order is preserved: when more than one item is scheduled for
                /// time T then at time T they will be run in the order that they were scheduled.
                inline worker create_worker(composite_subscription cs = composite_subscription()) const {
                        return inner->create_worker(cs);
                }
        };

        template<class Scheduler, class... ArgN>
        inline scheduler make_scheduler(ArgN&&... an) {
                return
scheduler(std::static_pointer_cast<scheduler_interface>(std::make_shared<Scheduler>(std::forward<ArgN>(an)...)));
        }

        inline scheduler make_scheduler(std::shared_ptr<scheduler_interface> si) {
                return scheduler(si);
        }

        class schedulable : public schedulable_base
        {
                typedef schedulable this_type;

                composite_subscription lifetime;
                weak_worker controller;
                action activity;
                bool scoped;
                composite_subscription::weak_subscription action_scope;

                struct detacher
                {
                        ~detacher()
                        {
                                if (that) {
                                        that->unsubscribe();
                                }
                        }
                        detacher(const this_type* that)
                                : that(that)
                        {
                        }
                        const this_type* that;
                };

                class recursed_scope_type
                {
                        mutable const recursed* requestor;

                        class exit_recursed_scope_type
                        {
                                const recursed_scope_type* that;
                        public:
                                ~exit_recursed_scope_type()
                                {
                                        that->requestor = nullptr;
                                }
                                exit_recursed_scope_type(const recursed_scope_type* that)
                                        : that(that)
                                {
                                }
                        };
                public:
                        recursed_scope_type()
                                : requestor(nullptr)
                        {
                        }
                        recursed_scope_type(const recursed_scope_type&)
                                : requestor(nullptr)
                        {
                                // does not aquire recursion scope
                        }
                        recursed_scope_type& operator=(const recursed_scope_type&)
                        {
                                // no change in recursion scope
                                return *this;
                        }
                        exit_recursed_scope_type reset(const recurse& r) const {
                                requestor = std::addressof(r.get_recursed());
                                return exit_recursed_scope_type(this);
                        }
                        bool is_recursed() const {
                                return !!requestor;
```

```
                }
                void operator()() const {
                        (*requestor)();
                }
        };
        recursed_scope_type recursed_scope;

public:
        typedef composite_subscription::weak_subscription weak_subscription;
        typedef scheduler_base::clock_type clock_type;

        ~schedulable()
        {
                if (scoped) {
                        controller.lock().remove(action_scope);
                }
        }
        schedulable()
                : scoped(false)
        {
        }

        /// action and worker share lifetime
        schedulable(worker q, action a)
                : lifetime(q.get_subscription())
                , controller(q)
                , activity(std::move(a))
                , scoped(false)
        {
        }
        /// action and worker have independent lifetimes
        schedulable(composite_subscription cs, worker q, action a)
                : lifetime(std::move(cs))
                , controller(q)
                , activity(std::move(a))
                , scoped(true)
                , action_scope(controller.lock().add(lifetime))
        {
        }
        /// inherit lifetimes
        schedulable(schedulable scbl, worker q, action a)
                : lifetime(scbl.get_subscription())
                , controller(q)
                , activity(std::move(a))
                , scoped(scbl.scoped)
                , action_scope(scbl.scoped ? controller.lock().add(lifetime) : weak_subscription())
        {
        }

        inline const composite_subscription& get_subscription() const {
                return lifetime;
        }
        inline composite_subscription& get_subscription() {
                return lifetime;
        }
        inline const worker get_worker() const {
                return controller.lock();
        }
        inline worker get_worker() {
                return controller.lock();
        }
        inline const action& get_action() const {
                return activity;
        }
        inline action& get_action() {
                return activity;
        }

        inline static schedulable empty(worker sc) {
                return schedulable(composite_subscription::empty(), sc, action::empty());
        }

        inline auto set_recursed(const recurse& r) const
                -> decltype(recursed_scope.reset(r)) {
                return      recursed_scope.reset(r);
        }

        // recursed
        //
        bool is_recursed() const {
                return recursed_scope.is_recursed();
        }
```

```
                                /// requests tail-recursion of the same action
                                /// this will exit the process if called when
                                /// is_recursed() is false.
                                /// Note: to improve perf it is not required
                                /// to call is_recursed() before calling this
                                /// operator. Context is sufficient. The schedulable
                                /// passed to the action by the scheduler will return
                                /// true from is_recursed()
                                inline void operator()() const {
                                        recursed_scope();
                                }

                                // composite_subscription
                                //
                                inline bool is_subscribed() const {
                                        return lifetime.is_subscribed();
                                }
                                inline weak_subscription add(subscription s) const {
                                        return lifetime.add(std::move(s));
                                }
                                template<class F>
                                auto add(F f) const
                                        -> typename std::enable_if<rxcpp::detail::is_unsubscribe_function<F>::value,
weak_subscription>::type {
                                        return lifetime.add(make_subscription(std::move(f)));
                                }
                                inline void remove(weak_subscription w) const {
                                        return lifetime.remove(std::move(w));
                                }
                                inline void clear() const {
                                        return lifetime.clear();
                                }
                                inline void unsubscribe() const {
                                        return lifetime.unsubscribe();
                                }

                                // scheduler
                                //
                                inline clock_type::time_point now() const {
                                        return controller.lock().now();
                                }
                                /// put this on the queue of the stored scheduler to run asap
                                inline void schedule() const {
                                        if (is_subscribed()) {
                                                get_worker().schedule(*this);
                                        }
                                }
                                /// put this on the queue of the stored scheduler to run at the specified time
                                inline void schedule(clock_type::time_point when) const {
                                        if (is_subscribed()) {
                                                get_worker().schedule(when, *this);
                                        }
                                }
                                /// put this on the queue of the stored scheduler to run after a delay from now
                                inline void schedule(clock_type::duration when) const {
                                        if (is_subscribed()) {
                                                get_worker().schedule(when, *this);
                                        }
                                }

                                // action
                                //
                                /// invokes the action
                                inline void operator()(const recurse& r) const {
                                        if (!is_subscribed()) {
                                                return;
                                        }
                                        detacher protect(this);
                                        activity(*this, r);
                                        protect.that = nullptr;
                                }
                        };

                        struct current_thread;

                        namespace detail {

                                class action_type
                                        : public std::enable_shared_from_this<action_type>
                                {
                                        typedef action_type this_type;
```

```
		public:
			typedef std::function<void(const schedulable&, const recurse&)> function_type;

		private:
			function_type f;

		public:
			action_type()
			{
			}

			action_type(function_type f)
				: f(std::move(f))
			{
			}

			inline void operator()(const schedulable& s, const recurse& r) {
				if (!f) {
					std::terminate();
				}
				f(s, r);
			}
		};

		class action_tailrecurser
			: public std::enable_shared_from_this<action_type>
		{
			typedef action_type this_type;

		public:
			typedef std::function<void(const schedulable&)> function_type;

		private:
			function_type f;

		public:
			action_tailrecurser()
			{
			}

			action_tailrecurser(function_type f)
				: f(std::move(f))
			{
			}

			inline void operator()(const schedulable& s, const recurse& r) {
				if (!f) {
					std::terminate();
				}
				trace_activity().action_enter(s);
				auto scope = s.set_recursed(r);
				while (s.is_subscribed()) {
					r.reset();
					f(s);
					if (!r.is_allowed() || !r.is_requested()) {
						if (r.is_requested()) {
							s.schedule();
						}
						break;
					}
					trace_activity().action_recurse(s);
				}
				trace_activity().action_return(s);
			}
		};
	}

	inline void action::operator()(const schedulable& s, const recurse& r) const {
		(*inner)(s, r);
	}

	inline action make_action_empty() {
		return action::empty();
	}

	template<class F>
	inline action make_action(F&& f) {
		static_assert(detail::is_action_function<F>::value, "action function must be void(schedulable)");
		auto fn = std::forward<F>(f);
		return action(std::make_shared<detail::action_type>(detail::action_tailrecurser(fn)));
	}
```

```
// copy
inline auto make_schedulable(
	const	schedulable& scbl)
	->	schedulable {
	return	schedulable(scbl);
}
// move
inline auto make_schedulable(
	schedulable&& scbl)
	->	schedulable {
	return	schedulable(std::move(scbl));
}

inline schedulable make_schedulable(worker sc, action a) {
	return schedulable(sc, a);
}
inline schedulable make_schedulable(worker sc, composite_subscription cs, action a) {
	return schedulable(cs, sc, a);
}

template<class F>
auto make_schedulable(worker sc, F&& f)
	-> typename std::enable_if<detail::is_action_function<F>::value, schedulable>::type {
	return schedulable(sc, make_action(std::forward<F>(f)));
}
template<class F>
auto make_schedulable(worker sc, composite_subscription cs, F&& f)
	-> typename std::enable_if<detail::is_action_function<F>::value, schedulable>::type {
	return schedulable(cs, sc, make_action(std::forward<F>(f)));
}
template<class F>
auto make_schedulable(schedulable scbl, composite_subscription cs, F&& f)
	-> typename std::enable_if<detail::is_action_function<F>::value, schedulable>::type {
	return schedulable(cs, scbl.get_worker(), make_action(std::forward<F>(f)));
}
template<class F>
auto make_schedulable(schedulable scbl, worker sc, F&& f)
	-> typename std::enable_if<detail::is_action_function<F>::value, schedulable>::type {
	return schedulable(scbl, sc, make_action(std::forward<F>(f)));
}
template<class F>
auto make_schedulable(schedulable scbl, F&& f)
	-> typename std::enable_if<detail::is_action_function<F>::value, schedulable>::type {
	return schedulable(scbl, scbl.get_worker(), make_action(std::forward<F>(f)));
}

inline auto make_schedulable(schedulable scbl, composite_subscription cs)
	-> schedulable {
	return schedulable(cs, scbl.get_worker(), scbl.get_action());
}
inline auto make_schedulable(schedulable scbl, worker sc, composite_subscription cs)
	-> schedulable {
	return schedulable(cs, sc, scbl.get_action());
}
inline auto make_schedulable(schedulable scbl, worker sc)
	-> schedulable {
	return schedulable(scbl, sc, scbl.get_action());
}

template<class Arg0, class... ArgN>
auto worker::schedule(Arg0&& a0, ArgN&&... an) const
	-> typename std::enable_if<
	(detail::is_action_function<Arg0>::value ||
	is_subscription<Arg0>::value) &&
	!is_schedulable<Arg0>::value>::type {
	auto scbl = make_schedulable(*this, std::forward<Arg0>(a0), std::forward<ArgN>(an)...);
	trace_activity().schedule_enter(*inner.get(), scbl);
	inner->schedule(std::move(scbl));
	trace_activity().schedule_return(*inner.get());
}
template<class... ArgN>
void worker::schedule_rebind(const schedulable& scbl, ArgN&&... an) const {
	auto rescbl = make_schedulable(scbl, *this, std::forward<ArgN>(an)...);
	trace_activity().schedule_enter(*inner.get(), rescbl);
	inner->schedule(std::move(rescbl));
	trace_activity().schedule_return(*inner.get());
}

template<class Arg0, class... ArgN>
auto worker::schedule(clock_type::time_point when, Arg0&& a0, ArgN&&... an) const
	-> typename std::enable_if<
	(detail::is_action_function<Arg0>::value ||
```

```
                    is_subscription<Arg0>::value) &&
                    !is_schedulable<Arg0>::value>::type {
                    auto scbl = make_schedulable(*this, std::forward<Arg0>(a0), std::forward<ArgN>(an)...);
                    trace_activity().schedule_when_enter(*inner.get(), when, scbl);
                    inner->schedule(when, std::move(scbl));
                    trace_activity().schedule_when_return(*inner.get());
            }
            template<class... ArgN>
            void worker::schedule_rebind(clock_type::time_point when, const schedulable& scbl, ArgN&&... an) const {
                    auto rescbl = make_schedulable(scbl, *this, std::forward<ArgN>(an)...);
                    trace_activity().schedule_when_enter(*inner.get(), when, rescbl);
                    inner->schedule(when, std::move(rescbl));
                    trace_activity().schedule_when_return(*inner.get());
            }

            template<class Arg0, class... ArgN>
            auto worker::schedule_periodically(clock_type::time_point initial, clock_type::duration period, Arg0&& a0, ArgN&&... an)
const
                    -> typename std::enable_if<
                    (detail::is_action_function<Arg0>::value ||
                    is_subscription<Arg0>::value) &&
                    !is_schedulable<Arg0>::value>::type {
                    schedule_periodically_rebind(initial, period, make_schedulable(*this, std::forward<Arg0>(a0),
std::forward<ArgN>(an)...));
            }
            template<class... ArgN>
            void worker::schedule_periodically_rebind(clock_type::time_point initial, clock_type::duration period, const schedulable&
scbl, ArgN&&... an) const {
                    auto keepAlive = *this;
                    auto target = std::make_shared<clock_type::time_point>(initial);
                    auto activity = make_schedulable(scbl, keepAlive, std::forward<ArgN>(an)...);
                    auto periodic = make_schedulable(
                            activity,
                            [keepAlive, target, period, activity](schedulable self) {
                            // any recursion requests will be pushed to the scheduler queue
                            recursion r(false);
                            // call action
                            activity(r.get_recurse());

                            // schedule next occurance (if the action took longer than 'period' target will be in the past)
                            *target += period;
                            self.schedule(*target);
                    });
                    trace_activity().schedule_when_enter(*inner.get(), *target, periodic);
                    inner->schedule(*target, periodic);
                    trace_activity().schedule_when_return(*inner.get());
            }

            namespace detail {

                    template<class TimePoint>
                    struct time_schedulable
                    {
                            typedef TimePoint time_point_type;

                            time_schedulable(TimePoint when, schedulable a)
                                    : when(when)
                                    , what(std::move(a))
                            {
                            }
                            TimePoint when;
                            schedulable what;
                    };


                    // Sorts time_schedulable items in priority order sorted
                    // on value of time_schedulable.when. Items with equal
                    // values for when are sorted in fifo order.
                    template<class TimePoint>
                    class schedulable_queue {
                    public:
                            typedef time_schedulable<TimePoint> item_type;
                            typedef std::pair<item_type, int64_t> elem_type;
                            typedef std::vector<elem_type> container_type;
                            typedef const item_type& const_reference;

                    private:
                            struct compare_elem
                            {
                                    bool operator()(const elem_type& lhs, const elem_type& rhs) const {
                                            if (lhs.first.when == rhs.first.when) {
                                                    return lhs.second > rhs.second;
```

```
                                                }
                                                else {
                                                        return lhs.first.when > rhs.first.when;
                                                }
                                        }
                                };

                                typedef std::priority_queue<
                                        elem_type,
                                        container_type,
                                        compare_elem
                                > queue_type;

                                queue_type q;

                                int64_t ordinal;
                        public:

                                schedulable_queue()
                                        : ordinal(0)
                                {
                                }

                                const_reference top() const {
                                        return q.top().first;
                                }

                                void pop() {
                                        q.pop();
                                }

                                bool empty() const {
                                        return q.empty();
                                }

                                void push(const item_type& value) {
                                        q.push(elem_type(value, ordinal++));
                                }

                                void push(item_type&& value) {
                                        q.push(elem_type(std::move(value), ordinal++));
                                }
                        };

                }

        }
        namespace rxsc = schedulers;

}

#if !defined(RXCPP_RX_SCHEDULER_CURRENT_THREAD_HPP)
#define RXCPP_RX_SCHEDULER_CURRENT_THREAD_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace schedulers {

                namespace detail {

                        struct action_queue
                        {
                                typedef action_queue this_type;

                                typedef scheduler_base::clock_type clock;
                                typedef time_schedulable<clock::time_point> item_type;

                        private:
                                typedef schedulable_queue<item_type::time_point_type> queue_item_time;

                        public:
                                struct current_thread_queue_type {
                                        std::shared_ptr<worker_interface> w;
                                        recursion r;
                                        queue_item_time q;
                                };

                        private:
#if defined(RXCPP_THREAD_LOCAL)
                                static current_thread_queue_type*& current_thread_queue() {
```

```
                static RXCPP_THREAD_LOCAL current_thread_queue_type* q;
                return q;
            }
#else
            static rxu::thread_local_storage<current_thread_queue_type>& current_thread_queue() {
                static rxu::thread_local_storage<current_thread_queue_type> q;
                return q;
            }
#endif

        public:

            static bool owned() {
                return !!current_thread_queue();
            }
            static const std::shared_ptr<worker_interface>& get_worker_interface() {
                return current_thread_queue()->w;
            }
            static recursion& get_recursion() {
                return current_thread_queue()->r;
            }
            static bool empty() {
                if (!current_thread_queue()) {
                    std::terminate();
                }
                return current_thread_queue()->q.empty();
            }
            static queue_item_time::const_reference top() {
                if (!current_thread_queue()) {
                    std::terminate();
                }
                return current_thread_queue()->q.top();
            }
            static void pop() {
                auto& state = current_thread_queue();
                if (!state) {
                    std::terminate();
                }
                state->q.pop();
                if (state->q.empty()) {
                    // allow recursion
                    state->r.reset(true);
                }
            }
            static void push(item_type item) {
                auto& state = current_thread_queue();
                if (!state) {
                    std::terminate();
                }
                if (!item.what.is_subscribed()) {
                    return;
                }
                state->q.push(std::move(item));
                // disallow recursion
                state->r.reset(false);
            }
            static std::shared_ptr<worker_interface> ensure(std::shared_ptr<worker_interface> w) {
                if (!!current_thread_queue()) {
                    std::terminate();
                }
                // create and publish new queue
                current_thread_queue() = new current_thread_queue_type();
                current_thread_queue()->w = w;
                return w;
            }
            static std::unique_ptr<current_thread_queue_type> create(std::shared_ptr<worker_interface> w) {
                std::unique_ptr<current_thread_queue_type> result(new current_thread_queue_type());
                result->w = std::move(w);
                return result;
            }
            static void set(current_thread_queue_type* q) {
                if (!!current_thread_queue()) {
                    std::terminate();
                }
                // publish new queue
                current_thread_queue() = q;
            }
            static void destroy(current_thread_queue_type* q) {
                delete q;
            }
            static void destroy() {
                if (!current_thread_queue()) {
```

```
                                                std::terminate();
                                        }
#if defined(RXCPP_THREAD_LOCAL)
                                        destroy(current_thread_queue());
#else
                                        destroy(current_thread_queue().get());
#endif
                                        current_thread_queue() = nullptr;
                                }
                        };

                }

                struct current_thread : public scheduler_interface
                {
                private:
                        typedef current_thread this_type;
                        current_thread(const this_type&);

                        typedef detail::action_queue queue_type;

                        struct derecurser : public worker_interface
                        {
                        private:
                                typedef current_thread this_type;
                                derecurser(const this_type&);
                        public:
                                derecurser()
                                {
                                }
                                virtual ~derecurser()
                                {
                                }

                                virtual clock_type::time_point now() const {
                                        return clock_type::now();
                                }

                                virtual void schedule(const schedulable& scbl) const {
                                        queue_type::push(queue_type::item_type(now(), scbl));
                                }

                                virtual void schedule(clock_type::time_point when, const schedulable& scbl) const {
                                        queue_type::push(queue_type::item_type(when, scbl));
                                }
                        };

                        struct current_worker : public worker_interface
                        {
                        private:
                                typedef current_thread this_type;
                                current_worker(const this_type&);
                        public:
                                current_worker()
                                {
                                }
                                virtual ~current_worker()
                                {
                                }

                                virtual clock_type::time_point now() const {
                                        return clock_type::now();
                                }

                                virtual void schedule(const schedulable& scbl) const {
                                        schedule(now(), scbl);
                                }

                                virtual void schedule(clock_type::time_point when, const schedulable& scbl) const {
                                        if (!scbl.is_subscribed()) {
                                                return;
                                        }

                                        {
                                                // check ownership
                                                if (queue_type::owned()) {
                                                        // already has an owner - delegate
                                                        queue_type::get_worker_interface()->schedule(when, scbl);
                                                        return;
                                                }

                                                // take ownership
```

```
                                                queue_type::ensure(std::make_shared<derecurser>());
                                        }
                                        // release ownership
                                        RXCPP_UNWIND_AUTO([]{
                                                queue_type::destroy();
                                        });

                                        const auto& recursor = queue_type::get_recursion().get_recurse();
                                        std::this_thread::sleep_until(when);
                                        if (scbl.is_subscribed()) {
                                                scbl(recursor);
                                        }
                                        if (queue_type::empty()) {
                                                return;
                                        }

                                        // loop until queue is empty
                                        for (
                                                auto next = queue_type::top().when;
                                                (std::this_thread::sleep_until(next), true);
                                        next = queue_type::top().when
                                                ) {
                                                auto what = queue_type::top().what;

                                                queue_type::pop();

                                                if (what.is_subscribed()) {
                                                        what(recursor);
                                                }

                                                if (queue_type::empty()) {
                                                        break;
                                                }
                                        }
                                }
                        };

                        std::shared_ptr<current_worker> wi;

                public:
                        current_thread()
                                : wi(std::make_shared<current_worker>())
                        {
                        }
                        virtual ~current_thread()
                        {
                        }

                        static bool is_schedule_required() { return !queue_type::owned(); }

                        inline bool is_tail_recursion_allowed() const {
                                return queue_type::empty();
                        }

                        virtual clock_type::time_point now() const {
                                return clock_type::now();
                        }

                        virtual worker create_worker(composite_subscription cs) const {
                                return worker(std::move(cs), wi);
                        }
                };

                inline const scheduler& make_current_thread() {
                        static scheduler instance = make_scheduler<current_thread>();
                        return instance;
                }

        }

}

#endif


#if !defined(RXCPP_RX_SCHEDULER_RUN_LOOP_HPP)
#define RXCPP_RX_SCHEDULER_RUN_LOOP_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {
```

```
namespace schedulers {

	namespace detail {

		struct run_loop_state : public std::enable_shared_from_this<run_loop_state>
		{
			typedef scheduler::clock_type clock_type;

			typedef detail::schedulable_queue<
				clock_type::time_point> queue_item_time;

			typedef queue_item_time::item_type item_type;
			typedef queue_item_time::const_reference const_reference_item_type;

			virtual ~run_loop_state()
			{
			}

			run_loop_state()
			{
			}

			composite_subscription lifetime;
			mutable std::mutex lock;
			mutable queue_item_time q;
			recursion r;
		};

	}


	struct run_loop_scheduler : public scheduler_interface
	{
	private:
		typedef run_loop_scheduler this_type;
		run_loop_scheduler(const this_type&);

		struct run_loop_worker : public worker_interface
		{
		private:
			typedef run_loop_worker this_type;

			run_loop_worker(const this_type&);

		public:
			std::weak_ptr<detail::run_loop_state> state;

			virtual ~run_loop_worker()
			{
			}

			explicit run_loop_worker(std::weak_ptr<detail::run_loop_state> ws)
				: state(ws)
			{
			}

			virtual clock_type::time_point now() const {
				return clock_type::now();
			}

			virtual void schedule(const schedulable& scbl) const {
				schedule(now(), scbl);
			}

			virtual void schedule(clock_type::time_point when, const schedulable& scbl) const {
				if (scbl.is_subscribed()) {
					auto st = state.lock();
					std::unique_lock<std::mutex> guard(st->lock);
					st->q.push(detail::run_loop_state::item_type(when, scbl));
					st->r.reset(false);
				}
			}
		};

		std::weak_ptr<detail::run_loop_state> state;

	public:
		explicit run_loop_scheduler(std::weak_ptr<detail::run_loop_state> ws)
			: state(ws)
		{
		}
		virtual ~run_loop_scheduler()
```

```
		{
		}

		virtual clock_type::time_point now() const {
			return clock_type::now();
		}

		virtual worker create_worker(composite_subscription cs) const {
			auto lifetime = state.lock()->lifetime;
			auto token = lifetime.add(cs);
			cs.add([=](){lifetime.remove(token); });
			return worker(cs, create_worker_interface());
		}

		std::shared_ptr<worker_interface> create_worker_interface() const {
			return std::make_shared<run_loop_worker>(state);
		}
};

class run_loop
{
private:
		typedef run_loop this_type;
		// don't allow this instance to copy/move since it owns current_thread queue
		// for the thread it is constructed on.
		run_loop(const this_type&);
		run_loop(this_type&&);

		typedef scheduler::clock_type clock_type;
		typedef detail::action_queue queue_type;

		typedef detail::run_loop_state::item_type item_type;
		typedef detail::run_loop_state::const_reference_item_type const_reference_item_type;

		std::shared_ptr<detail::run_loop_state> state;
		std::shared_ptr<run_loop_scheduler> sc;

public:
		run_loop()
			: state(std::make_shared<detail::run_loop_state>())
			, sc(std::make_shared<run_loop_scheduler>(state))
		{
			// take ownership so that the current_thread scheduler
			// uses the same queue on this thread
			queue_type::ensure(sc->create_worker_interface());
		}
		~run_loop()
		{
			state->lifetime.unsubscribe();

			std::unique_lock<std::mutex> guard(state->lock);

			// release ownership
			queue_type::destroy();

			auto expired = std::move(state->q);
			if (!state->q.empty()) std::terminate();
		}

		clock_type::time_point now() const {
			return clock_type::now();
		}

		composite_subscription get_subscription() const {
			return state->lifetime;
		}

		bool empty() const {
			return state->q.empty();
		}

		const_reference_item_type peek() const {
			return state->q.top();
		}

		void dispatch() const {
			std::unique_lock<std::mutex> guard(state->lock);
			if (state->q.empty()) {
				return;
			}
			auto& peek = state->q.top();
			if (!peek.what.is_subscribed()) {
```

```
                                                state->q.pop();
                                                return;
                                        }
                                        if (clock_type::now() < peek.when) {
                                                return;
                                        }
                                        auto what = peek.what;
                                        state->q.pop();
                                        state->r.reset(state->q.empty());
                                        guard.unlock();
                                        what(state->r.get_recurse());
                                }

                                scheduler get_scheduler() const {
                                        return make_scheduler(sc);
                                }
                        };

                        inline scheduler make_run_loop(const run_loop& r) {
                                return r.get_scheduler();
                        }

                }

}

#endif


#if !defined(RXCPP_RX_SCHEDULER_NEW_THREAD_HPP)
#define RXCPP_RX_SCHEDULER_NEW_THREAD_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace schedulers {

                typedef std::function<std::thread(std::function<void()>)> thread_factory;

                struct new_thread : public scheduler_interface
                {
                private:
                        typedef new_thread this_type;
                        new_thread(const this_type&);

                        struct new_worker : public worker_interface
                        {
                        private:
                                typedef new_worker this_type;

                                typedef detail::action_queue queue_type;

                                new_worker(const this_type&);

                                struct new_worker_state : public std::enable_shared_from_this<new_worker_state>
                                {
                                        typedef detail::schedulable_queue<
                                        typename clock_type::time_point> queue_item_time;

                                        typedef queue_item_time::item_type item_type;

                                        virtual ~new_worker_state()
                                        {
                                                std::unique_lock<std::mutex> guard(lock);
                                                if (worker.joinable() && worker.get_id() != std::this_thread::get_id()) {
                                                        lifetime.unsubscribe();
                                                        guard.unlock();
                                                        worker.join();
                                                }
                                                else {
                                                        lifetime.unsubscribe();
                                                        worker.detach();
                                                }
                                        }

                                        explicit new_worker_state(composite_subscription cs)
                                                : lifetime(cs)
                                        {
                                        }

                                        composite_subscription lifetime;
```

```
                mutable std::mutex lock;
                mutable std::condition_variable wake;
                mutable queue_item_time q;
                std::thread worker;
                recursion r;
        };

        std::shared_ptr<new_worker_state> state;

public:
        virtual ~new_worker()
        {
        }

        explicit new_worker(std::shared_ptr<new_worker_state> ws)
                : state(ws)
        {
        }

        new_worker(composite_subscription cs, thread_factory& tf)
                : state(std::make_shared<new_worker_state>(cs))
        {
                auto keepAlive = state;

                state->lifetime.add([keepAlive](){
                        std::unique_lock<std::mutex> guard(keepAlive->lock);
                        auto expired = std::move(keepAlive->q);
                        if (!keepAlive->q.empty()) std::terminate();
                        keepAlive->wake.notify_one();
                });

                state->worker = tf([keepAlive](){

                        // take ownership
                        queue_type::ensure(std::make_shared<new_worker>(keepAlive));
                        // release ownership
                        RXCPP_UNWIND_AUTO([]{
                                queue_type::destroy();
                        });

                        for (;;) {
                                std::unique_lock<std::mutex> guard(keepAlive->lock);
                                if (keepAlive->q.empty()) {
                                        keepAlive->wake.wait(guard, [keepAlive](){
                                                return !keepAlive->lifetime.is_subscribed() ||
!keepAlive->q.empty();
                                        });
                                }
                                if (!keepAlive->lifetime.is_subscribed()) {
                                        break;
                                }
                                auto& peek = keepAlive->q.top();
                                if (!peek.what.is_subscribed()) {
                                        keepAlive->q.pop();
                                        continue;
                                }
                                if (clock_type::now() < peek.when) {
                                        keepAlive->wake.wait_until(guard, peek.when);
                                        continue;
                                }
                                auto what = peek.what;
                                keepAlive->q.pop();
                                keepAlive->r.reset(keepAlive->q.empty());
                                guard.unlock();
                                what(keepAlive->r.get_recurse());
                        }
                });
        }

        virtual clock_type::time_point now() const {
                return clock_type::now();
        }

        virtual void schedule(const schedulable& scbl) const {
                schedule(now(), scbl);
        }

        virtual void schedule(clock_type::time_point when, const schedulable& scbl) const {
                if (scbl.is_subscribed()) {
                        std::unique_lock<std::mutex> guard(state->lock);
                        state->q.push(new_worker_state::item_type(when, scbl));
                        state->r.reset(false);
```

```
                                        }
                                        state->wake.notify_one();
                                }
                        };

                        mutable thread_factory factory;

                public:
                        new_thread()
                                : factory([](std::function<void()> start){
                                return std::thread(std::move(start));
                        })
                        {
                        }
                        explicit new_thread(thread_factory tf)
                                : factory(tf)
                        {
                        }
                        virtual ~new_thread()
                        {
                        }

                        virtual clock_type::time_point now() const {
                                return clock_type::now();
                        }

                        virtual worker create_worker(composite_subscription cs) const {
                                return worker(cs, std::make_shared<new_worker>(cs, factory));
                        }
                };

                inline scheduler make_new_thread() {
                        static scheduler instance = make_scheduler<new_thread>();
                        return instance;
                }
                inline scheduler make_new_thread(thread_factory tf) {
                        return make_scheduler<new_thread>(tf);
                }

        }

}

#endif


#if !defined(RXCPP_RX_SCHEDULER_EVENT_LOOP_HPP)
#define RXCPP_RX_SCHEDULER_EVENT_LOOP_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace schedulers {

                struct event_loop : public scheduler_interface
                {
                private:
                        typedef event_loop this_type;
                        event_loop(const this_type&);

                        struct loop_worker : public worker_interface
                        {
                        private:
                                typedef loop_worker this_type;
                                loop_worker(const this_type&);

                                typedef detail::schedulable_queue<
                                        typename clock_type::time_point> queue_item_time;

                                typedef queue_item_time::item_type item_type;

                                composite_subscription lifetime;
                                worker controller;

                        public:
                                virtual ~loop_worker()
                                {
                                }
                                loop_worker(composite_subscription cs, worker w)
                                        : lifetime(cs)
                                        , controller(w)
```

```
                                        {
                                        }

                                        virtual clock_type::time_point now() const {
                                                return clock_type::now();
                                        }

                                        virtual void schedule(const schedulable& scbl) const {
                                                controller.schedule(lifetime, scbl.get_action());
                                        }

                                        virtual void schedule(clock_type::time_point when, const schedulable& scbl) const {
                                                controller.schedule(when, lifetime, scbl.get_action());
                                        }
                                };

                                mutable thread_factory factory;
                                scheduler newthread;
                                mutable std::atomic<std::size_t> count;
                                std::vector<worker> loops;

                        public:
                                event_loop()
                                        : factory([](std::function<void()> start){
                                        return std::thread(std::move(start));
                                })
                                        , newthread(make_new_thread())
                                        , count(0)
                                {
                                        auto remaining = std::max(std::thread::hardware_concurrency(), unsigned(4));
                                        while (remaining--) {
                                                loops.push_back(newthread.create_worker());
                                        }
                                }
                                explicit event_loop(thread_factory tf)
                                        : factory(tf)
                                        , newthread(make_new_thread(tf))
                                        , count(0)
                                {
                                        auto remaining = std::max(std::thread::hardware_concurrency(), unsigned(4));
                                        while (remaining--) {
                                                loops.push_back(newthread.create_worker());
                                        }
                                }
                                virtual ~event_loop()
                                {
                                }

                                virtual clock_type::time_point now() const {
                                        return clock_type::now();
                                }

                                virtual worker create_worker(composite_subscription cs) const {
                                        return worker(cs, std::make_shared<loop_worker>(cs, loops[++count % loops.size()]));
                                }
                        };

                        inline scheduler make_event_loop() {
                                static scheduler instance = make_scheduler<event_loop>();
                                return instance;
                        }
                        inline scheduler make_event_loop(thread_factory tf) {
                                return make_scheduler<event_loop>(tf);
                        }

                }

}

#endif


#if !defined(RXCPP_RX_SCHEDULER_IMMEDIATE_HPP)
#define RXCPP_RX_SCHEDULER_IMMEDIATE_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace schedulers {

                struct immediate : public scheduler_interface
```

```
		{
		private:
			typedef immediate this_type;
			immediate(const this_type&);

			struct immediate_worker : public worker_interface
			{
			private:
				typedef immediate_worker this_type;
				immediate_worker(const this_type&);
			public:
				virtual ~immediate_worker()
				{
				}
				immediate_worker()
				{
				}

				virtual clock_type::time_point now() const {
					return clock_type::now();
				}

				virtual void schedule(const schedulable& scbl) const {
					if (scbl.is_subscribed()) {
						// allow recursion
						recursion r(true);
						scbl(r.get_recurse());
					}
				}

				virtual void schedule(clock_type::time_point when, const schedulable& scbl) const {
					std::this_thread::sleep_until(when);
					if (scbl.is_subscribed()) {
						// allow recursion
						recursion r(true);
						scbl(r.get_recurse());
					}
				}
			};

			std::shared_ptr<immediate_worker> wi;

		public:
			immediate()
				: wi(std::make_shared<immediate_worker>())
			{
			}
			virtual ~immediate()
			{
			}

			virtual clock_type::time_point now() const {
				return clock_type::now();
			}

			virtual worker create_worker(composite_subscription cs) const {
				return worker(std::move(cs), wi);
			}
		};

		inline const scheduler& make_immediate() {
			static scheduler instance = make_scheduler<immediate>();
			return instance;
		}

	}

}

#endif


#if !defined(RXCPP_RX_SCHEDULER_VIRTUAL_TIME_HPP)
#define RXCPP_RX_SCHEDULER_VIRTUAL_TIME_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace schedulers {

		namespace detail {
```

```
template<class Absolute, class Relative>
struct virtual_time_base : std::enable_shared_from_this<virtual_time_base<Absolute, Relative>>
{
private:
	typedef virtual_time_base<Absolute, Relative> this_type;
	virtual_time_base(const virtual_time_base&);

	mutable bool isenabled;

public:
	typedef Absolute absolute;
	typedef Relative relative;

	virtual ~virtual_time_base()
	{
	}

protected:
	virtual_time_base()
		: isenabled(false)
		, clock_now(0)
	{
	}
	explicit virtual_time_base(absolute initialClock)
		: isenabled(false)
		, clock_now(initialClock)
	{
	}

	mutable absolute clock_now;

	typedef time_schedulable<long> item_type;

	virtual absolute add(absolute, relative) const = 0;

	virtual typename scheduler_base::clock_type::time_point to_time_point(absolute) const = 0;
	virtual relative to_relative(typename scheduler_base::clock_type::duration) const = 0;

	virtual item_type top() const = 0;
	virtual void pop() const = 0;
	virtual bool empty() const = 0;

public:

	virtual void schedule_absolute(absolute, const schedulable&) const = 0;

	virtual void schedule_relative(relative when, const schedulable& a) const {
		auto at = add(clock_now, when);
		return schedule_absolute(at, a);
	}

	bool is_enabled() const { return isenabled; }
	absolute clock() const { return clock_now; }

	void start() const
	{
		if (!isenabled) {
			isenabled = true;
			rxsc::recursion r;
			r.reset(false);
			while (!empty() && isenabled) {
				auto next = top();
				pop();
				if (next.what.is_subscribed()) {
					if (next.when > clock_now) {
						clock_now = next.when;
					}
					next.what(r.get_recurse());
				}
			}
			isenabled = false;
		}
	}

	void stop() const
	{
		isenabled = false;
	}

	void advance_to(absolute time) const
	{
```

```
                        if (time < clock_now) {
                                std::terminate();
                        }

                        if (time == clock_now) {
                                return;
                        }

                        if (!isenabled) {
                                isenabled = true;
                                rxsc::recursion r;
                                while (!empty() && isenabled) {
                                        auto next = top();
                                        if (next.when <= time) {
                                                pop();
                                                if (!next.what.is_subscribed()) {
                                                        continue;
                                                }
                                                if (next.when > clock_now) {
                                                        clock_now = next.when;
                                                }
                                                next.what(r.get_recurse());
                                        }
                                        else {
                                                break;
                                        }
                                }
                                isenabled = false;
                                clock_now = time;
                        }
                        else {
                                std::terminate();
                        }
                }

                void advance_by(relative time) const
                {
                        auto dt = add(clock_now, time);

                        if (dt < clock_now) {
                                std::terminate();
                        }

                        if (dt == clock_now) {
                                return;
                        }

                        if (!isenabled) {
                                advance_to(dt);
                        }
                        else {
                                std::terminate();
                        }
                }

                void sleep(relative time) const
                {
                        auto dt = add(clock_now, time);

                        if (dt < clock_now) {
                                std::terminate();
                        }

                        clock_now = dt;
                }

        };

}

template<class Absolute, class Relative>
struct virtual_time : public detail::virtual_time_base<Absolute, Relative>
{
        typedef detail::virtual_time_base<Absolute, Relative> base;

        typedef typename base::item_type item_type;

        typedef detail::schedulable_queue<
                typename item_type::time_point_type> queue_item_time;

        mutable queue_item_time q;
```

```
                public:
                        virtual ~virtual_time()
                        {
                        }

                protected:
                        virtual_time()
                        {
                        }
                        explicit virtual_time(typename base::absolute initialClock)
                                : base(initialClock)
                        {
                        }

                        virtual item_type top() const {
                                return q.top();
                        }
                        virtual void pop() const {
                                q.pop();
                        }
                        virtual bool empty() const {
                                return q.empty();
                        }

                        using base::schedule_absolute;
                        using base::schedule_relative;

                        virtual void schedule_absolute(typename base::absolute when, const schedulable& a) const
                        {
                                // use a separate subscription here so that a's subscription is not affected
                                auto run = make_schedulable(
                                        a.get_worker(),
                                        composite_subscription(),
                                        [a](const schedulable& scbl) {
                                        rxsc::recursion r;
                                        r.reset(false);
                                        if (scbl.is_subscribed()) {
                                                scbl.unsubscribe(); // unsubscribe() run, not a;
                                                a(r.get_recurse());
                                        }
                                });
                                q.push(item_type(when, run));
                        }
                };



        }

}

#endif


#if !defined(RXCPP_RX_SCHEDULER_SAME_WORKER_HPP)
#define RXCPP_RX_SCHEDULER_SAME_WORKER_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace schedulers {

                struct same_worker : public scheduler_interface
                {
                private:
                        typedef same_worker this_type;
                        same_worker(const this_type&);

                        rxsc::worker controller;

                public:
                        explicit same_worker(rxsc::worker w)
                                : controller(std::move(w))
                        {
                        }
                        virtual ~same_worker()
                        {
                        }

                        virtual clock_type::time_point now() const {
                                return controller.now();
```

```
                }

                virtual worker create_worker(composite_subscription cs) const {
                    // use different lifetime
                    auto inner_lifetime = controller.get_subscription();
                    auto token = inner_lifetime.add(cs);
                    cs.add([inner_lifetime, token](){inner_lifetime.remove(token); });
                    return worker(cs, controller);
                }
            };

            inline scheduler make_same_worker(rxsc::worker w) {
                return make_scheduler<same_worker>(std::move(w));
            }

        }

}

#endif


#endif


#if !defined(RXCPP_RX_SUBSCRIBER_HPP)
#define RXCPP_RX_SUBSCRIBER_HPP

//_include "rx-includes.hpp"

namespace rxcpp {

    template<class T>
    struct subscriber_base : public observer_base<T>, public subscription_base
    {
        typedef tag_subscriber subscriber_tag;
    };

    /*!
    \brief binds an observer that consumes values with a composite_subscription that controls lifetime.

    \ingroup group-core

    */
    template<class T, class Observer = observer<T>>
    class subscriber : public subscriber_base<T>
    {
        static_assert(!is_subscriber<Observer>::value, "not allowed to nest subscribers");
        static_assert(is_observer<Observer>::value, "subscriber must contain an observer<T, ...>");
        typedef subscriber<T, Observer> this_type;
        typedef rxu::decay_t<Observer> observer_type;

        composite_subscription lifetime;
        observer_type destination;
        trace_id id;

        struct nextdetacher
        {
            ~nextdetacher()
            {
                trace_activity().on_next_return(*that);
                if (do_unsubscribe) {
                    that->unsubscribe();
                }
            }
            nextdetacher(const this_type* that)
                : that(that)
                , do_unsubscribe(true)
            {
            }
            template<class U>
            void operator()(U u) {
                trace_activity().on_next_enter(*that, u);
                try {
                    that->destination.on_next(std::move(u));
                    do_unsubscribe = false;
                }
                catch (...) {
                    auto ex = std::current_exception();
                    trace_activity().on_error_enter(*that, ex);
                    that->destination.on_error(std::move(ex));
                    trace_activity().on_error_return(*that);
```

```
                                }
                        }
                        const this_type* that;
                        volatile bool do_unsubscribe;
                };

                struct errordetacher
                {
                        ~errordetacher()
                        {
                                trace_activity().on_error_return(*that);
                                that->unsubscribe();
                        }
                        errordetacher(const this_type* that)
                                : that(that)
                        {
                        }
                        inline void operator()(std::exception_ptr ex) {
                                trace_activity().on_error_enter(*that, ex);
                                that->destination.on_error(std::move(ex));
                        }
                        const this_type* that;
                };

                struct completeddetacher
                {
                        ~completeddetacher()
                        {
                                trace_activity().on_completed_return(*that);
                                that->unsubscribe();
                        }
                        completeddetacher(const this_type* that)
                                : that(that)
                        {
                        }
                        inline void operator()() {
                                trace_activity().on_completed_enter(*that);
                                that->destination.on_completed();
                        }
                        const this_type* that;
                };

                subscriber();
        public:
                typedef typename composite_subscription::weak_subscription weak_subscription;

                subscriber(const this_type& o)
                        : lifetime(o.lifetime)
                        , destination(o.destination)
                        , id(o.id)
                {
                }
                subscriber(this_type&& o)
                        : lifetime(std::move(o.lifetime))
                        , destination(std::move(o.destination))
                        , id(std::move(o.id))
                {
                }

                template<class U, class O>
                friend class subscriber;

                template<class O>
                subscriber(
                        const subscriber<T, O>& o,
                        typename std::enable_if<
                        !std::is_same<O, observer<T>>::value &&
                        std::is_same<Observer, observer<T>>::value, void**>::type = nullptr)
                        : lifetime(o.lifetime)
                        , destination(o.destination.as_dynamic())
                        , id(o.id)
                {
                }

                template<class U>
                subscriber(trace_id id, composite_subscription cs, U&& o)
                        : lifetime(std::move(cs))
                        , destination(std::forward<U>(o))
                        , id(std::move(id))
                {
                        static_assert(!is_subscriber<U>::value, "cannot nest subscribers");
                        static_assert(is_observer<U>::value, "must pass observer to subscriber");
```

```
                trace_activity().create_subscriber(*this);
        }

        this_type& operator=(this_type o) {
                lifetime = std::move(o.lifetime);
                destination = std::move(o.destination);
                id = std::move(o.id);
                return *this;
        }

        const observer_type& get_observer() const {
                return destination;
        }
        observer_type& get_observer() {
                return destination;
        }
        const composite_subscription& get_subscription() const {
                return lifetime;
        }
        composite_subscription& get_subscription() {
                return lifetime;
        }
        trace_id get_id() const {
                return id;
        }

        subscriber<T> as_dynamic() const {
                return subscriber<T>(id, lifetime, destination.as_dynamic());
        }

        // observer
        //
        template<class V>
        void on_next(V&& v) const {
                if (!is_subscribed()) {
                        return;
                }
                nextdetacher protect(this);
                protect(std::forward<V>(v));
        }
        void on_error(std::exception_ptr e) const {
                if (!is_subscribed()) {
                        return;
                }
                errordetacher protect(this);
                protect(std::move(e));
        }
        void on_completed() const {
                if (!is_subscribed()) {
                        return;
                }
                completeddetacher protect(this);
                protect();
        }

        // composite_subscription
        //
        bool is_subscribed() const {
                return lifetime.is_subscribed();
        }
        weak_subscription add(subscription s) const {
                return lifetime.add(std::move(s));
        }
        template<class F>
        auto add(F f) const
                -> typename std::enable_if<detail::is_unsubscribe_function<F>::value, weak_subscription>::type {
                return lifetime.add(make_subscription(std::move(f)));
        }
        void remove(weak_subscription w) const {
                return lifetime.remove(std::move(w));
        }
        void clear() const {
                return lifetime.clear();
        }
        void unsubscribe() const {
                return lifetime.unsubscribe();
        }

};

template<class T, class Observer>
auto make_subscriber(
```

```
                subscriber<T, Observer> o)
                ->      subscriber<T, Observer> {
                return  subscriber<T, Observer>(std::move(o));
        }

        // observer
        //

        template<class T>
        auto make_subscriber()
                -> typename std::enable_if<
                detail::is_on_next_of<T, detail::OnNextEmpty<T>>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, detail::OnNextEmpty<T>> >> ::type {
                return  subscriber<T, observer<T, detail::stateless_observer_tag,
detail::OnNextEmpty<T>>>(trace_id::make_next_id_subscriber(), composite_subscription(),
                        observer<T, detail::stateless_observer_tag, detail::OnNextEmpty<T>>(detail::OnNextEmpty<T>()));
        }

        template<class T, class I>
        auto make_subscriber(
                const                   observer<T, I>& o)
                ->      subscriber<T, observer<T, I>> {
                return  subscriber<T, observer<T, I>>(trace_id::make_next_id_subscriber(), composite_subscription(), o);
        }
        template<class T, class Observer>
        auto make_subscriber(const Observer& o)
                -> typename std::enable_if<
                is_observer<Observer>::value &&
                !is_subscriber<Observer>::value,
                subscriber<T, Observer >> ::type {
                return  subscriber<T, Observer>(trace_id::make_next_id_subscriber(), composite_subscription(), o);
        }
        template<class T, class Observer>
        auto make_subscriber(const Observer& o)
                -> typename std::enable_if<
                !detail::is_on_next_of<T, Observer>::value &&
                !is_subscriber<Observer>::value &&
                !is_subscription<Observer>::value &&
                !is_observer<Observer>::value,
                subscriber<T, observer<T, Observer>>>::type {
                return  subscriber<T, observer<T, Observer>>(trace_id::make_next_id_subscriber(), composite_subscription(), o);
        }
        template<class T, class OnNext>
        auto make_subscriber(const OnNext& on)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>>::type {
                return  subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>(trace_id::make_next_id_subscriber(),
composite_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext>(on));
        }
        template<class T, class OnNext, class OnError>
        auto make_subscriber(const OnNext& on, const OnError& oe)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>>::type {
                return  subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>(trace_id::make_next_id_subscriber(),
composite_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext, OnError>(on, oe));
        }
        template<class T, class OnNext, class OnCompleted>
        auto make_subscriber(const OnNext& on, const OnCompleted& oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_completed<OnCompleted>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>>>::type {
                return  subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty,
OnCompleted>>(trace_id::make_next_id_subscriber(), composite_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>(on,
detail::OnErrorEmpty(), oc));
        }
        template<class T, class OnNext, class OnError, class OnCompleted>
        auto make_subscriber(const OnNext& on, const OnError& oe, const OnCompleted& oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value &&
                detail::is_on_completed<OnCompleted>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>>>::type {
                return  subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError,
OnCompleted>>(trace_id::make_next_id_subscriber(), composite_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>(on, oe, oc));
```

```
	}

	// explicit lifetime
	//

	template<class T>
	auto make_subscriber(const composite_subscription& cs)
		->	subscriber<T, observer<T, detail::stateless_observer_tag, detail::OnNextEmpty<T>>> {
		return	subscriber<T, observer<T, detail::stateless_observer_tag,
detail::OnNextEmpty<T>>>(trace_id::make_next_id_subscriber(), cs,
				observer<T, detail::stateless_observer_tag, detail::OnNextEmpty<T>>(detail::OnNextEmpty<T>()));
	}

	template<class T, class I>
	auto make_subscriber(const composite_subscription& cs,
		const		observer<T, I>& o)
		->	subscriber<T, observer<T, I>> {
		return	subscriber<T, observer<T, I>>(trace_id::make_next_id_subscriber(), cs, o);
	}
	template<class T, class I>
	auto make_subscriber(const composite_subscription& cs,
		const		subscriber<T, I>& s)
		->	subscriber<T, I> {
		return	subscriber<T, I>(trace_id::make_next_id_subscriber(), cs, s.get_observer());
	}
	template<class T, class Observer>
	auto make_subscriber(const composite_subscription& cs, const Observer& o)
		-> typename std::enable_if<
		!is_subscriber<Observer>::value &&
		is_observer<Observer>::value,
		subscriber<T, Observer >> ::type {
		return	subscriber<T, Observer>(trace_id::make_next_id_subscriber(), cs, o);
	}
	template<class T, class Observer>
	auto make_subscriber(const composite_subscription& cs, const Observer& o)
		-> typename std::enable_if<
		!detail::is_on_next_of<T, Observer>::value &&
		!is_subscriber<Observer>::value &&
		!is_subscription<Observer>::value &&
		!is_observer<Observer>::value,
		subscriber<T, observer<T, Observer>>>::type {
		return	subscriber<T, observer<T, Observer>>(trace_id::make_next_id_subscriber(), cs, make_observer<T>(o));
	}
	template<class T, class OnNext>
	auto make_subscriber(const composite_subscription& cs, const OnNext& on)
		-> typename std::enable_if<
		detail::is_on_next_of<T, OnNext>::value,
		subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>>::type {
		return	subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>(trace_id::make_next_id_subscriber(), cs,
				observer<T, detail::stateless_observer_tag, OnNext>(on));
	}
	template<class T, class OnNext, class OnError>
	auto make_subscriber(const composite_subscription& cs, const OnNext& on, const OnError& oe)
		-> typename std::enable_if<
		detail::is_on_next_of<T, OnNext>::value &&
		detail::is_on_error<OnError>::value,
		subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>>::type {
		return	subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>(trace_id::make_next_id_subscriber(),
cs,
				observer<T, detail::stateless_observer_tag, OnNext, OnError>(on, oe));
	}
	template<class T, class OnNext, class OnCompleted>
	auto make_subscriber(const composite_subscription& cs, const OnNext& on, const OnCompleted& oc)
		-> typename std::enable_if<
		detail::is_on_next_of<T, OnNext>::value &&
		detail::is_on_completed<OnCompleted>::value,
		subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>>>::type {
		return	subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty,
OnCompleted>>(trace_id::make_next_id_subscriber(), cs,
				observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>(on,
detail::OnErrorEmpty(), oc));
	}
	template<class T, class OnNext, class OnError, class OnCompleted>
	auto make_subscriber(const composite_subscription& cs, const OnNext& on, const OnError& oe, const OnCompleted& oc)
		-> typename std::enable_if<
		detail::is_on_next_of<T, OnNext>::value &&
		detail::is_on_error<OnError>::value &&
		detail::is_on_completed<OnCompleted>::value,
		subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>>>::type {
		return	subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError,
OnCompleted>>(trace_id::make_next_id_subscriber(), cs,
				observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>(on, oe, oc));
```

```
        }

        // explicit id
        //

        template<class T>
        auto make_subscriber(trace_id id)
                ->      subscriber<T, observer<T, detail::stateless_observer_tag, detail::OnNextEmpty<T>>> {
                return  subscriber<T, observer<T, detail::stateless_observer_tag, detail::OnNextEmpty<T>>>(std::move(id),
composite_subscription(),
                                observer<T, detail::stateless_observer_tag, detail::OnNextEmpty<T>>(detail::OnNextEmpty<T>()));
        }

        template<class T>
        auto make_subscriber(trace_id id, const composite_subscription& cs)
                ->      subscriber<T, observer<T, detail::stateless_observer_tag, detail::OnNextEmpty<T>>> {
                return  subscriber<T, observer<T, detail::stateless_observer_tag, detail::OnNextEmpty<T>>>(std::move(id), cs,
                                observer<T, detail::stateless_observer_tag, detail::OnNextEmpty<T>>(detail::OnNextEmpty<T>()));
        }

        template<class T, class I>
        auto make_subscriber(trace_id id,
                const                   observer<T, I>& o)
                ->      subscriber<T, observer<T, I>> {
                return  subscriber<T, observer<T, I>>(std::move(id), composite_subscription(), o);
        }
        template<class T, class I>
        auto make_subscriber(trace_id id, const composite_subscription& cs,
                const                   observer<T, I>& o)
                ->      subscriber<T, observer<T, I>> {
                return  subscriber<T, observer<T, I>>(std::move(id), cs, o);
        }
        template<class T, class Observer>
        auto make_subscriber(trace_id id, const Observer& o)
                -> typename std::enable_if<
                is_observer<Observer>::value,
                subscriber<T, Observer >> ::type {
                return  subscriber<T, Observer>(std::move(id), composite_subscription(), o);
        }
        template<class T, class Observer>
        auto make_subscriber(trace_id id, const composite_subscription& cs, const Observer& o)
                -> typename std::enable_if<
                is_observer<Observer>::value,
                subscriber<T, Observer >> ::type {
                return  subscriber<T, Observer>(std::move(id), cs, o);
        }
        template<class T, class Observer>
        auto make_subscriber(trace_id id, const Observer& o)
                -> typename std::enable_if<
                !detail::is_on_next_of<T, Observer>::value &&
                !is_subscriber<Observer>::value &&
                !is_subscription<Observer>::value &&
                !is_observer<Observer>::value,
                subscriber<T, observer<T, Observer>>>::type {
                return  subscriber<T, observer<T, Observer>>(std::move(id), composite_subscription(), o);
        }
        template<class T, class Observer>
        auto make_subscriber(trace_id id, const composite_subscription& cs, const Observer& o)
                -> typename std::enable_if<
                !detail::is_on_next_of<T, Observer>::value &&
                !is_subscriber<Observer>::value &&
                !is_subscription<Observer>::value &&
                !is_observer<Observer>::value,
                subscriber<T, observer<T, Observer>>>::type {
                return  subscriber<T, observer<T, Observer>>(std::move(id), cs, o);
        }
        template<class T, class OnNext>
        auto make_subscriber(trace_id id, const OnNext& on)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>>::type {
                return  subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>(std::move(id), composite_subscription(),
                                observer<T, detail::stateless_observer_tag, OnNext>(on));
        }
        template<class T, class OnNext>
        auto make_subscriber(trace_id id, const composite_subscription& cs, const OnNext& on)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>>::type {
                return  subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>(std::move(id), cs,
                                observer<T, detail::stateless_observer_tag, OnNext>(on));
        }
```

```
        template<class T, class OnNext, class OnError>
        auto make_subscriber(trace_id id, const OnNext& on, const OnError& oe)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>>::type {
                return  subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>(std::move(id),
composite_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext, OnError>(on, oe));
        }
        template<class T, class OnNext, class OnError>
        auto make_subscriber(trace_id id, const composite_subscription& cs, const OnNext& on, const OnError& oe)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>>::type {
                return  subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>(std::move(id), cs,
                        observer<T, detail::stateless_observer_tag, OnNext, OnError>(on, oe));
        }
        template<class T, class OnNext, class OnCompleted>
        auto make_subscriber(trace_id id, const OnNext& on, const OnCompleted& oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_completed<OnCompleted>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>>>::type {
                return  subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty,
OnCompleted>>(std::move(id), composite_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>(on,
detail::OnErrorEmpty(), oc));
        }
        template<class T, class OnNext, class OnCompleted>
        auto make_subscriber(trace_id id, const composite_subscription& cs, const OnNext& on, const OnCompleted& oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_completed<OnCompleted>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>>>::type {
                return  subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty,
OnCompleted>>(std::move(id), cs,
                        observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>(on,
detail::OnErrorEmpty(), oc));
        }
        template<class T, class OnNext, class OnError, class OnCompleted>
        auto make_subscriber(trace_id id, const OnNext& on, const OnError& oe, const OnCompleted& oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value &&
                detail::is_on_completed<OnCompleted>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>>>::type {
                return  subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>>(std::move(id),
composite_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>(on, oe, oc));
        }
        template<class T, class OnNext, class OnError, class OnCompleted>
        auto make_subscriber(trace_id id, const composite_subscription& cs, const OnNext& on, const OnError& oe, const OnCompleted&
oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value &&
                detail::is_on_completed<OnCompleted>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>>>::type {
                return  subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>>(std::move(id), cs,
                        observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>(on, oe, oc));
        }

        // chain defaults from subscriber
        //

        template<class T, class OtherT, class OtherObserver, class I>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr,
                const                   observer<T, I>& o)
                ->      subscriber<T, observer<T, I>> {
                auto r = subscriber<T, observer<T, I>>(trace_id::make_next_id_subscriber(), scbr.get_subscription(), o);
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class I>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, trace_id id,
                const                   observer<T, I>& o)
                ->      subscriber<T, observer<T, I>> {
                auto r = subscriber<T, observer<T, I>>(std::move(id), scbr.get_subscription(), o);
                trace_activity().connect(r, scbr);
                return r;
```

```
        }
        template<class T, class OtherT, class OtherObserver, class Observer>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, trace_id id, const Observer& o)
                -> typename std::enable_if<
                is_observer<Observer>::value,
                subscriber<T, Observer >> ::type {
                auto r = subscriber<T, Observer>(std::move(id), scbr.get_subscription(), o);
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class Observer>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, const Observer& o)
                -> typename std::enable_if<
                !is_subscription<Observer>::value &&
                is_observer<Observer>::value,
                subscriber<T, Observer >> ::type {
                auto r = subscriber<T, Observer>(trace_id::make_next_id_subscriber(), scbr.get_subscription(), o);
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class Observer>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, const Observer& o)
                -> typename std::enable_if<
                !detail::is_on_next_of<T, Observer>::value &&
                !is_subscriber<Observer>::value &&
                !is_subscription<Observer>::value &&
                !is_observer<Observer>::value,
                subscriber<T, observer<T, Observer>>>::type {
                auto r = subscriber<T, observer<T, Observer>>(trace_id::make_next_id_subscriber(), scbr.get_subscription(),
make_observer<T>(o));
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class Observer>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, trace_id id, const Observer& o)
                -> typename std::enable_if<
                !detail::is_on_next_of<T, Observer>::value &&
                !is_subscriber<Observer>::value &&
                !is_subscription<Observer>::value &&
                !is_observer<Observer>::value,
                subscriber<T, observer<T, Observer>>>::type {
                auto r = subscriber<T, observer<T, Observer>>(std::move(id), scbr.get_subscription(), o);
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, const OnNext& on)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>(trace_id::make_next_id_subscriber(),
scbr.get_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext>(on));
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, trace_id id, const OnNext& on)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>(std::move(id), scbr.get_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext>(on));
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext, class OnError>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, const OnNext& on, const OnError& oe)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>(trace_id::make_next_id_subscriber(),
scbr.get_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext, OnError>(on, oe));
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext, class OnError>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, trace_id id, const OnNext& on, const OnError& oe)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
```

```
                detail::is_on_error<OnError>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>(std::move(id),
scbr.get_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext, OnError>(on, oe));
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext, class OnCompleted>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, const OnNext& on, const OnCompleted& oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_completed<OnCompleted>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty,
OnCompleted>>(trace_id::make_next_id_subscriber(), scbr.get_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>(on,
detail::OnErrorEmpty(), oc));
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext, class OnCompleted>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, trace_id id, const OnNext& on, const OnCompleted& oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_completed<OnCompleted>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty,
OnCompleted>>(std::move(id), scbr.get_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>(on,
detail::OnErrorEmpty(), oc));
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext, class OnError, class OnCompleted>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, const OnNext& on, const OnError& oe, const OnCompleted&
oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value &&
                detail::is_on_completed<OnCompleted>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError,
OnCompleted>>(trace_id::make_next_id_subscriber(), scbr.get_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>(on, oe, oc));
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext, class OnError, class OnCompleted>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, trace_id id, const OnNext& on, const OnError& oe, const
OnCompleted& oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value &&
                detail::is_on_completed<OnCompleted>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>>(std::move(id),
scbr.get_subscription(),
                        observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>(on, oe, oc));
                trace_activity().connect(r, scbr);
                return r;
        }

        template<class T, class OtherT, class OtherObserver, class I>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>&, const composite_subscription& cs,
                const          observer<T, I>& o)
                ->      subscriber<T, observer<T, I>> {
                return  subscriber<T, observer<T, I>>(trace_id::make_next_id_subscriber(), cs, o);
        }
        template<class T, class OtherT, class OtherObserver, class I>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>&, trace_id id, const composite_subscription& cs,
                const          observer<T, I>& o)
                ->      subscriber<T, observer<T, I>> {
                return  subscriber<T, observer<T, I>>(std::move(id), cs, o);
        }
        template<class T, class OtherT, class OtherObserver, class Observer>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, const composite_subscription& cs, const Observer& o)
                -> typename std::enable_if<
                is_observer<Observer>::value,
                subscriber<T, Observer >> ::type {
                auto r = subscriber<T, Observer>(trace_id::make_next_id_subscriber(), cs, o);
                trace_activity().connect(r, scbr);
```

```
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class Observer>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, trace_id id, const composite_subscription& cs, const
Observer& o)
                -> typename std::enable_if<
                is_observer<Observer>::value,
                subscriber<T, Observer >> ::type {
                auto r = subscriber<T, Observer>(std::move(id), cs, o);
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class Observer>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, const composite_subscription& cs, const Observer& o)
                -> typename std::enable_if<
                !detail::is_on_next_of<T, Observer>::value &&
                !is_subscriber<Observer>::value &&
                !is_subscription<Observer>::value &&
                !is_observer<Observer>::value,
                subscriber<T, observer<T, Observer>>>::type {
                auto r = subscriber<T, observer<T, Observer>>(trace_id::make_next_id_subscriber(), cs, o);
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class Observer>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, trace_id id, const composite_subscription& cs, const
Observer& o)
                -> typename std::enable_if<
                !detail::is_on_next_of<T, Observer>::value &&
                !is_subscriber<Observer>::value &&
                !is_subscription<Observer>::value &&
                !is_observer<Observer>::value,
                subscriber<T, observer<T, Observer>>>::type {
                auto r = subscriber<T, observer<T, Observer>>(std::move(id), cs, o);
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, const composite_subscription& cs, const OnNext& on)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>(trace_id::make_next_id_subscriber(), cs,
                        observer<T, detail::stateless_observer_tag, OnNext>(on));
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, trace_id id, const composite_subscription& cs, const
OnNext& on)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext>>(std::move(id), cs,
                        observer<T, detail::stateless_observer_tag, OnNext>(on));
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext, class OnError>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, const composite_subscription& cs, const OnNext& on, const
OnError& oe)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>(trace_id::make_next_id_subscriber(),
cs,
                        observer<T, detail::stateless_observer_tag, OnNext, OnError>(on, oe));
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext, class OnError>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, trace_id id, const composite_subscription& cs, const
OnNext& on, const OnError& oe)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError>>(std::move(id), cs,
                        observer<T, detail::stateless_observer_tag, OnNext, OnError>(on, oe));
                trace_activity().connect(r, scbr);
                return r;
```

```
        }
        template<class T, class OtherT, class OtherObserver, class OnNext, class OnCompleted>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, const composite_subscription& cs, const OnNext& on, const
OnCompleted& oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_completed<OnCompleted>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty,
OnCompleted>>(trace_id::make_next_id_subscriber(), cs,
                        observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>(on,
detail::OnErrorEmpty(), oc));
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext, class OnCompleted>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, trace_id id, const composite_subscription& cs, const
OnNext& on, const OnCompleted& oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_completed<OnCompleted>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty,
OnCompleted>>(std::move(id), cs,
                        observer<T, detail::stateless_observer_tag, OnNext, detail::OnErrorEmpty, OnCompleted>(on,
detail::OnErrorEmpty(), oc));
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext, class OnError, class OnCompleted>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, const composite_subscription& cs, const OnNext& on, const
OnError& oe, const OnCompleted& oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value &&
                detail::is_on_completed<OnCompleted>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError,
OnCompleted>>(trace_id::make_next_id_subscriber(), cs,
                        observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>(on, oe, oc));
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class OtherT, class OtherObserver, class OnNext, class OnError, class OnCompleted>
        auto make_subscriber(const subscriber<OtherT, OtherObserver>& scbr, trace_id id, const composite_subscription& cs, const
OnNext& on, const OnError& oe, const OnCompleted& oc)
                -> typename std::enable_if<
                detail::is_on_next_of<T, OnNext>::value &&
                detail::is_on_error<OnError>::value &&
                detail::is_on_completed<OnCompleted>::value,
                subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>>>::type {
                auto r = subscriber<T, observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>>(std::move(id), cs,
                        observer<T, detail::stateless_observer_tag, OnNext, OnError, OnCompleted>(on, oe, oc));
                trace_activity().connect(r, scbr);
                return r;
        }

        template<class T, class Observer>
        auto make_subscriber(const subscriber<T, Observer>& scbr, const composite_subscription& cs)
                ->      subscriber<T, Observer> {
                auto r = subscriber<T, Observer>(scbr.get_id(), cs, scbr.get_observer());
                trace_activity().connect(r, scbr);
                return r;
        }
        template<class T, class Observer>
        auto make_subscriber(const subscriber<T, Observer>& scbr, trace_id id, const composite_subscription& cs)
                ->      subscriber<T, Observer> {
                auto r = subscriber<T, Observer>(std::move(id), cs, scbr.get_observer());
                trace_activity().connect(r, scbr);
                return r;
        }

        template<class T, class Observer>
        auto make_subscriber(const subscriber<T, Observer>& scbr, trace_id id)
                ->      subscriber<T, Observer> {
                auto r = subscriber<T, Observer>(std::move(id), scbr.get_subscription(), scbr.get_observer());
                trace_activity().connect(r, scbr);
                return r;
        }

}
```

```
#endif


#if !defined(RXCPP_RX_NOTIFICATION_HPP)
#define RXCPP_RX_NOTIFICATION_HPP

//_include "rx-includes.hpp"

namespace rxcpp {

	namespace notifications {

		class subscription
		{
			long s;
			long u;

		public:
			explicit inline subscription(long s)
				: s(s), u(std::numeric_limits<long>::max()) {
			}
			inline subscription(long s, long u)
				: s(s), u(u) {
			}
			inline long subscribe() const {
				return s;
			}
			inline long unsubscribe() const {
				return u;
			}
		};

		inline bool operator == (subscription lhs, subscription rhs) {
			return lhs.subscribe() == rhs.subscribe() && lhs.unsubscribe() == rhs.unsubscribe();
		}

		inline std::ostream& operator<< (std::ostream& out, const subscription& s) {
			out << s.subscribe() << "-" << s.unsubscribe();
			return out;
		}

		namespace detail {

			template<typename T>
			struct notification_base
				: public std::enable_shared_from_this<notification_base<T>>
			{
				typedef subscriber<T> observer_type;
				typedef std::shared_ptr<notification_base<T>> type;

				virtual ~notification_base() {}

				virtual void out(std::ostream& out) const = 0;
				virtual bool equals(const type& other) const = 0;
				virtual void accept(const observer_type& o) const = 0;
			};

			template<class T>
			std::ostream& operator<< (std::ostream& out, const std::vector<T>& v);

			template<class T>
			auto to_stream(std::ostream& os, const T& t, int, int)
				-> decltype(os << t) {
				return      os << t;
			}

#if RXCPP_USE_RTTI
			template<class T>
			std::ostream& to_stream(std::ostream& os, const T&, int, ...) {
				return os << "< " << typeid(T).name() << " does not support ostream>";
			}
#endif

			template<class T>
			std::ostream& to_stream(std::ostream& os, const T&, ...) {
				return os << "<the value does not support ostream>";
			}

			template<class T>
			inline std::ostream& ostreamvector(std::ostream& os, const std::vector<T>& v) {
				os << "[";
				bool doemit = false;
```

```
                for (auto& i : v) {
                        if (doemit) {
                                os << ", ";
                        }
                        else {
                                doemit = true;
                        }
                        to_stream(os, i, 0, 0);
                }
                os << "]";
                return os;
        }

        template<class T>
        inline std::ostream& operator<< (std::ostream& os, const std::vector<T>& v) {
                return ostreamvector(os, v);
        }

        template<class T>
        auto equals(const T& lhs, const T& rhs, int)
                -> decltype(bool(lhs == rhs)) {
                return lhs == rhs;
        }

        template<class T>
        bool equals(const T&, const T&, ...) {
                throw std::runtime_error("value does not support equality tests");
                return false;
        }

}

template<typename T>
struct notification
{
        typedef typename detail::notification_base<T>::type type;
        typedef typename detail::notification_base<T>::observer_type observer_type;

private:
        typedef detail::notification_base<T> base;

        struct on_next_notification : public base {
                on_next_notification(T value) : value(std::move(value)) {
                }
                on_next_notification(const on_next_notification& o) : value(o.value) {}
                on_next_notification(const on_next_notification&& o) : value(std::move(o.value)) {}
                on_next_notification& operator=(on_next_notification o) { value = std::move(o.value); return *this; }
                virtual void out(std::ostream& os) const {
                        os << "on_next( ";
                        detail::to_stream(os, value, 0, 0);
                        os << ")";
                }
                virtual bool equals(const typename base::type& other) const {
                        bool result = false;
                        other->accept(make_subscriber<T>(make_observer_dynamic<T>([this, &result](T v) {
                                result = detail::equals(this->value, v, 0);
                        })));
                        return result;
                }
                virtual void accept(const typename base::observer_type& o) const {
                        o.on_next(value);
                }
                const T value;
        };

        struct on_error_notification : public base {
                on_error_notification(std::exception_ptr ep) : ep(ep) {
                }
                on_error_notification(const on_error_notification& o) : ep(o.ep) {}
                on_error_notification(const on_error_notification&& o) : ep(std::move(o.ep)) {}
                on_error_notification& operator=(on_error_notification o) { ep = std::move(o.ep); return *this; }
                virtual void out(std::ostream& os) const {
                        os << "on_error(";
                        try {
                                std::rethrow_exception(ep);
                        }
                        catch (const std::exception& e) {
                                os << e.what();
                        }
                        catch (...) {
                                os << "<not derived from std::exception>";
                        }
```

```
                                        os << ")";
                                }
                                virtual bool equals(const typename base::type& other) const {
                                        bool result = false;
                                        // not trying to compare exceptions
                                        other->accept(make_subscriber<T>(make_observer_dynamic<T>([](T){},
[&result](std::exception_ptr){
                                                result = true;
                                        })));
                                        return result;
                                }
                                virtual void accept(const typename base::observer_type& o) const {
                                        o.on_error(ep);
                                }
                                const std::exception_ptr ep;
                        };

                        struct on_completed_notification : public base {
                                on_completed_notification() {
                                }
                                virtual void out(std::ostream& os) const {
                                        os << "on_completed()";
                                }
                                virtual bool equals(const typename base::type& other) const {
                                        bool result = false;
                                        other->accept(make_subscriber<T>(make_observer_dynamic<T>([](T){}, [&result](){
                                                result = true;
                                        })));
                                        return result;
                                }
                                virtual void accept(const typename base::observer_type& o) const {
                                        o.on_completed();
                                }
                        };

                        struct exception_tag {};

                        template<typename Exception>
                        static
                                type make_on_error(exception_tag&&, Exception&& e) {
                                        std::exception_ptr ep;
                                        try {
                                                throw std::forward<Exception>(e);
                                        }
                                        catch (...) {
                                                ep = std::current_exception();
                                        }
                                        return std::make_shared<on_error_notification>(ep);
                                }

                        struct exception_ptr_tag {};

                        static
                                type make_on_error(exception_ptr_tag&&, std::exception_ptr ep) {
                                        return std::make_shared<on_error_notification>(ep);
                                }

                public:
                        template<typename U>
                        static type on_next(U value) {
                                return std::make_shared<on_next_notification>(std::move(value));
                        }

                        static type on_completed() {
                                return std::make_shared<on_completed_notification>();
                        }

                        template<typename Exception>
                        static type on_error(Exception&& e) {
                                return make_on_error(typename std::conditional<
                                        std::is_same<rxu::decay_t<Exception>, std::exception_ptr>::value,
                                        exception_ptr_tag, exception_tag>::type(),
                                        std::forward<Exception>(e));
                        }
                };

                template<class T>
                bool operator == (const std::shared_ptr<detail::notification_base<T>>& lhs, const
std::shared_ptr<detail::notification_base<T>>& rhs) {
                        if (!lhs && !rhs) { return true; }
                        if (!lhs || !rhs) { return false; }
                        return lhs->equals(rhs);
```

```
		}

		template<class T>
		std::ostream& operator<< (std::ostream& os, const std::shared_ptr<detail::notification_base<T>>& n) {
			n->out(os);
			return os;
		}


		template<class T>
		class recorded
		{
			long t;
			T v;
		public:
			recorded(long t, T v)
				: t(t), v(v) {
			}
			long time() const {
				return t;
			}
			const T& value() const {
				return v;
			}
		};

		template<class T>
		bool operator == (recorded<T> lhs, recorded<T> rhs) {
			return lhs.time() == rhs.time() && lhs.value() == rhs.value();
		}

		template<class T>
		std::ostream& operator<< (std::ostream& out, const recorded<T>& r) {
			out << "@" << r.time() << "-" << r.value();
			return out;
		}

	}
	namespace rxn = notifications;

}

inline std::ostream& operator<< (std::ostream& out, const std::vector<rxcpp::notifications::subscription>& vs) {
	return rxcpp::notifications::detail::ostreamvector(out, vs);
}
template<class T>
inline std::ostream& operator<< (std::ostream& out, const std::vector<rxcpp::notifications::recorded<T>>& vr) {
	return rxcpp::notifications::detail::ostreamvector(out, vr);
}

#endif


#if !defined(RXCPP_RX_COORDINATION_HPP)
#define RXCPP_RX_COORDINATION_HPP

//_include "rx-includes.hpp"

namespace rxcpp {

	struct tag_coordinator {};
	struct coordinator_base { typedef tag_coordinator coordinator_tag; };

	template<class T, class C = rxu::types_checked>
	struct is_coordinator : public std::false_type {};

	template<class T>
	struct is_coordinator<T, typename rxu::types_checked_from<typename T::coordinator_tag>::type>
		: public std::is_convertible<typename T::coordinator_tag*, tag_coordinator*>{};

	struct tag_coordination {};
	struct coordination_base { typedef tag_coordination coordination_tag; };

	namespace detail {

		template<class T, class C = rxu::types_checked>
		struct is_coordination : public std::false_type {};

		template<class T>
		struct is_coordination<T, typename rxu::types_checked_from<typename T::coordination_tag>::type>
			: public std::is_convertible<typename T::coordination_tag*, tag_coordination*>{};
```

```
}

template<class T, class Decayed = rxu::decay_t<T>>
struct is_coordination : detail::is_coordination<Decayed>
{
};

template<class Coordination, class DecayedCoordination = rxu::decay_t<Coordination>>
using coordination_tag_t = typename DecayedCoordination::coordination_tag;

template<class Input>
class coordinator : public coordinator_base
{
public:
	typedef Input input_type;

private:
	struct not_supported { typedef not_supported type; };

	template<class Observable>
	struct get_observable
	{
		typedef decltype((*(input_type*)nullptr).in((*(Observable*)nullptr))) type;
	};

	template<class Subscriber>
	struct get_subscriber
	{
		typedef decltype((*(input_type*)nullptr).out((*(Subscriber*)nullptr))) type;
	};

	template<class F>
	struct get_action_function
	{
		typedef decltype((*(input_type*)nullptr).act((*(F*)nullptr))) type;
	};

public:
	input_type input;

	template<class T>
	struct get
	{
		typedef typename std::conditional<
		rxsc::detail::is_action_function<T>::value, get_action_function<T>, typename std::conditional<
		is_observable<T>::value, get_observable<T>, typename std::conditional<
		is_subscriber<T>::value, get_subscriber<T>, not_supported>::type>::type>::type::type type;
	};

	coordinator(Input i) : input(i) {}

	rxsc::worker get_worker() const {
		return input.get_worker();
	}
	rxsc::scheduler get_scheduler() const {
		return input.get_scheduler();
	}

	template<class Observable>
	auto in(Observable o) const
		-> typename get_observable<Observable>::type {
		return input.in(std::move(o));
		static_assert(is_observable<Observable>::value, "can only synchronize observables");
	}

	template<class Subscriber>
	auto out(Subscriber s) const
		-> typename get_subscriber<Subscriber>::type {
		return input.out(std::move(s));
		static_assert(is_subscriber<Subscriber>::value, "can only synchronize subscribers");
	}

	template<class F>
	auto act(F f) const
		-> typename get_action_function<F>::type {
		return input.act(std::move(f));
		static_assert(rxsc::detail::is_action_function<F>::value, "can only synchronize action functions");
	}
};

class identity_one_worker : public coordination_base
{
```

```
		rxsc::scheduler factory;

		class input_type
		{
			rxsc::worker controller;
			rxsc::scheduler factory;
		public:
			explicit input_type(rxsc::worker w)
				: controller(w)
				, factory(rxsc::make_same_worker(w))
			{
			}
			inline rxsc::worker get_worker() const {
				return controller;
			}
			inline rxsc::scheduler get_scheduler() const {
				return factory;
			}
			inline rxsc::scheduler::clock_type::time_point now() const {
				return factory.now();
			}
			template<class Observable>
			auto in(Observable o) const
				-> Observable {
				return o;
			}
			template<class Subscriber>
			auto out(Subscriber s) const
				-> Subscriber {
				return s;
			}
			template<class F>
			auto act(F f) const
				-> F {
				return f;
			}
		};

	public:

		explicit identity_one_worker(rxsc::scheduler sc) : factory(sc) {}

		typedef coordinator<input_type> coordinator_type;

		inline rxsc::scheduler::clock_type::time_point now() const {
			return factory.now();
		}

		inline coordinator_type create_coordinator(composite_subscription cs = composite_subscription()) const {
			auto w = factory.create_worker(std::move(cs));
			return coordinator_type(input_type(std::move(w)));
		}
	};

	inline identity_one_worker identity_immediate() {
		static identity_one_worker r(rxsc::make_immediate());
		return r;
	}

	inline identity_one_worker identity_current_thread() {
		static identity_one_worker r(rxsc::make_current_thread());
		return r;
	}

	inline identity_one_worker identity_same_worker(rxsc::worker w) {
		return identity_one_worker(rxsc::make_same_worker(w));
	}

	class serialize_one_worker : public coordination_base
	{
		rxsc::scheduler factory;

		template<class F>
		struct serialize_action
		{
			F dest;
			std::shared_ptr<std::mutex> lock;
			serialize_action(F d, std::shared_ptr<std::mutex> m)
				: dest(std::move(d))
				, lock(std::move(m))
			{
				if (!lock) {
```

```
                                        std::terminate();
                                }
                        }
                        auto operator()(const rxsc::schedulable& scbl) const
                                -> decltype(dest(scbl)) {
                                std::unique_lock<std::mutex> guard(*lock);
                                return dest(scbl);
                        }
        };

        template<class Observer>
        struct serialize_observer
        {
                typedef serialize_observer<Observer> this_type;
                typedef rxu::decay_t<Observer> dest_type;
                typedef typename dest_type::value_type value_type;
                typedef observer<value_type, this_type> observer_type;
                dest_type dest;
                std::shared_ptr<std::mutex> lock;

                serialize_observer(dest_type d, std::shared_ptr<std::mutex> m)
                        : dest(std::move(d))
                        , lock(std::move(m))
                {
                        if (!lock) {
                                std::terminate();
                        }
                }
                void on_next(value_type v) const {
                        std::unique_lock<std::mutex> guard(*lock);
                        dest.on_next(v);
                }
                void on_error(std::exception_ptr e) const {
                        std::unique_lock<std::mutex> guard(*lock);
                        dest.on_error(e);
                }
                void on_completed() const {
                        std::unique_lock<std::mutex> guard(*lock);
                        dest.on_completed();
                }

                template<class Subscriber>
                static subscriber<value_type, observer_type> make(const Subscriber& s, std::shared_ptr<std::mutex> m) {
                        return make_subscriber<value_type>(s, observer_type(this_type(s.get_observer(), std::move(m))));
                }
        };

        class input_type
        {
                rxsc::worker controller;
                rxsc::scheduler factory;
                std::shared_ptr<std::mutex> lock;
        public:
                explicit input_type(rxsc::worker w, std::shared_ptr<std::mutex> m)
                        : controller(w)
                        , factory(rxsc::make_same_worker(w))
                        , lock(std::move(m))
                {
                }
                inline rxsc::worker get_worker() const {
                        return controller;
                }
                inline rxsc::scheduler get_scheduler() const {
                        return factory;
                }
                inline rxsc::scheduler::clock_type::time_point now() const {
                        return factory.now();
                }
                template<class Observable>
                auto in(Observable o) const
                        -> Observable {
                        return o;
                }
                template<class Subscriber>
                auto out(const Subscriber& s) const
                        -> decltype(serialize_observer<decltype(s.get_observer())>::make(s, lock)) {
                        return      serialize_observer<decltype(s.get_observer())>::make(s, lock);
                }
                template<class F>
                auto act(F f) const
                        ->      serialize_action<F> {
                        return  serialize_action<F>(std::move(f), lock);
```

```
                        }
                };

        public:

                explicit serialize_one_worker(rxsc::scheduler sc) : factory(sc) {}

                typedef coordinator<input_type> coordinator_type;

                inline rxsc::scheduler::clock_type::time_point now() const {
                        return factory.now();
                }

                inline coordinator_type create_coordinator(composite_subscription cs = composite_subscription()) const {
                        auto w = factory.create_worker(std::move(cs));
                        std::shared_ptr<std::mutex> lock = std::make_shared<std::mutex>();
                        return coordinator_type(input_type(std::move(w), std::move(lock)));
                }
        };

        inline serialize_one_worker serialize_event_loop() {
                static serialize_one_worker r(rxsc::make_event_loop());
                return r;
        }

        inline serialize_one_worker serialize_new_thread() {
                static serialize_one_worker r(rxsc::make_new_thread());
                return r;
        }

        inline serialize_one_worker serialize_same_worker(rxsc::worker w) {
                return serialize_one_worker(rxsc::make_same_worker(w));
        }

}

#endif



#if !defined(RXCPP_RX_SOURCES_HPP)
#define RXCPP_RX_SOURCES_HPP

//_include "rx-includes.hpp"

namespace rxcpp {

        namespace sources {

                struct tag_source {};
                template<class T>
                struct source_base
                {
                        typedef T value_type;
                        typedef tag_source source_tag;
                };
                template<class T>
                class is_source
                {
                        template<class C>
                        static typename C::source_tag* check(int);
                        template<class C>
                        static void check(...);
                public:
                        static const bool value = std::is_convertible<decltype(check<rxu::decay_t<T>>(0)), tag_source*>::value;
                };

        }
        namespace rxs = sources;

}

#if !defined(RXCPP_SOURCES_RX_CREATE_HPP)
#define RXCPP_SOURCES_RX_CREATE_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace sources {

                namespace detail {
```

```
                template<class T, class OnSubscribe>
                struct create : public source_base<T>
                {
                        typedef create<T, OnSubscribe> this_type;

                        typedef rxu::decay_t<OnSubscribe> on_subscribe_type;

                        on_subscribe_type on_subscribe_function;

                        create(on_subscribe_type os)
                                : on_subscribe_function(std::move(os))
                        {
                        }

                        template<class Subscriber>
                        void on_subscribe(Subscriber o) const {

                                on_exception(
                                        [&](){
                                        this->on_subscribe_function(o);
                                        return true;
                                },
                                        o);
                        }
                };

        }

        template<class T, class OnSubscribe>
        auto create(OnSubscribe os)
                ->      observable<T, detail::create<T, OnSubscribe>> {
                return  observable<T, detail::create<T, OnSubscribe>>(
                        detail::create<T, OnSubscribe>(std::move(os)));
        }

    }

}

#endif


#if !defined(RXCPP_SOURCES_RX_RANGE_HPP)
#define RXCPP_SOURCES_RX_RANGE_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

    namespace sources {

        namespace detail {

                template<class T, class Coordination>
                struct range : public source_base<T>
                {
                        typedef rxu::decay_t<Coordination> coordination_type;
                        typedef typename coordination_type::coordinator_type coordinator_type;

                        struct range_state_type
                        {
                                range_state_type(T f, T l, std::ptrdiff_t s, coordination_type cn)
                                : next(f)
                                , last(l)
                                , step(s)
                                , coordination(std::move(cn))
                                {
                                }
                                mutable T next;
                                T last;
                                std::ptrdiff_t step;
                                coordination_type coordination;
                        };
                        range_state_type initial;
                        range(T f, T l, std::ptrdiff_t s, coordination_type cn)
                                : initial(f, l, s, std::move(cn))
                        {
                        }
                        template<class Subscriber>
                        void on_subscribe(Subscriber o) const {
                                static_assert(is_subscriber<Subscriber>::value, "subscribe must be passed a subscriber");
```

```
                                        // creates a worker whose lifetime is the same as this subscription
                                        auto coordinator = initial.coordination.create_coordinator(o.get_subscription());

                                        auto controller = coordinator.get_worker();

                                        auto state = initial;

                                        auto producer = [=](const rxsc::schedulable& self){
                                                auto& dest = o;
                                                if (!dest.is_subscribed()) {
                                                        // terminate loop
                                                        return;
                                                }

                                                // send next value
                                                dest.on_next(state.next);
                                                if (!dest.is_subscribed()) {
                                                        // terminate loop
                                                        return;
                                                }

                                                if (std::abs(state.last - state.next) < std::abs(state.step)) {
                                                        if (state.last != state.next) {
                                                                dest.on_next(state.last);
                                                        }
                                                        dest.on_completed();
                                                        // o is unsubscribed
                                                        return;
                                                }
                                                state.next = static_cast<T>(state.step + state.next);

                                                // tail recurse this same action to continue loop
                                                self();
                                        };

                                        auto selectedProducer = on_exception(
                                                [&](){return coordinator.act(producer); },
                                                o);
                                        if (selectedProducer.empty()) {
                                                return;
                                        }

                                        controller.schedule(selectedProducer.get());
                                }
                        };

                }

                template<class T>
                auto range(T first = 0, T last = std::numeric_limits<T>::max(), std::ptrdiff_t step = 1)
                        ->      observable<T, detail::range<T, identity_one_worker>> {
                        return  observable<T, detail::range<T, identity_one_worker>>(
                                detail::range<T, identity_one_worker>(first, last, step, identity_current_thread()));
                }
                template<class T, class Coordination>
                auto range(T first, T last, std::ptrdiff_t step, Coordination cn)
                        ->      observable<T, detail::range<T, Coordination>> {
                        return  observable<T, detail::range<T, Coordination>>(
                                detail::range<T, Coordination>(first, last, step, std::move(cn)));
                }
                template<class T, class Coordination>
                auto range(T first, T last, Coordination cn)
                        -> typename std::enable_if<is_coordination<Coordination>::value,
                        observable<T, detail::range<T, Coordination>>>::type {
                        return  observable<T, detail::range<T, Coordination>>(
                                detail::range<T, Coordination>(first, last, 1, std::move(cn)));
                }
                template<class T, class Coordination>
                auto range(T first, Coordination cn)
                        -> typename std::enable_if<is_coordination<Coordination>::value,
                        observable<T, detail::range<T, Coordination>>>::type {
                        return  observable<T, detail::range<T, Coordination>>(
                                detail::range<T, Coordination>(first, std::numeric_limits<T>::max(), 1, std::move(cn)));
                }

        }

}

#endif
```

```
#if !defined(RXCPP_SOURCES_RX_ITERATE_HPP)
#define RXCPP_SOURCES_RX_ITERATE_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace sources {

		namespace detail {

			template<class Collection>
			struct is_iterable
			{
				typedef rxu::decay_t<Collection> collection_type;

				struct not_void {};
				template<class CC>
				static auto check(int) -> decltype(std::begin(*(CC*)nullptr));
				template<class CC>
				static not_void check(...);

				static const bool value = !std::is_same<decltype(check<collection_type>(0)), not_void>::value;
			};

			template<class Collection>
			struct iterate_traits
			{
				typedef rxu::decay_t<Collection> collection_type;
				typedef decltype(std::begin(*(collection_type*)nullptr)) iterator_type;
				typedef rxu::value_type_t<std::iterator_traits<iterator_type>> value_type;
			};

			template<class Collection, class Coordination>
			struct iterate : public source_base<rxu::value_type_t<iterate_traits<Collection>>>
			{
				typedef iterate<Collection, Coordination> this_type;
				typedef iterate_traits<Collection> traits;

				typedef rxu::decay_t<Coordination> coordination_type;
				typedef typename coordination_type::coordinator_type coordinator_type;

				typedef typename traits::collection_type collection_type;
				typedef typename traits::iterator_type iterator_type;

				struct iterate_initial_type
				{
					iterate_initial_type(collection_type c, coordination_type cn)
					: collection(std::move(c))
					, coordination(std::move(cn))
					{
					}
					collection_type collection;
					coordination_type coordination;
				};
				iterate_initial_type initial;

				iterate(collection_type c, coordination_type cn)
					: initial(std::move(c), std::move(cn))
				{
				}
				template<class Subscriber>
				void on_subscribe(Subscriber o) const {
					static_assert(is_subscriber<Subscriber>::value, "subscribe must be passed a subscriber");

					typedef typename coordinator_type::template get<Subscriber>::type output_type;

					struct iterate_state_type
						: public iterate_initial_type
					{
						iterate_state_type(const iterate_initial_type& i, output_type o)
						: iterate_initial_type(i)
						, cursor(std::begin(iterate_initial_type::collection))
						, end(std::end(iterate_initial_type::collection))
						, out(std::move(o))
						{
						}
						iterate_state_type(const iterate_state_type& o)
							: iterate_initial_type(o)
							, cursor(std::begin(iterate_initial_type::collection))
							, end(std::end(iterate_initial_type::collection))
```

```
                                                                , out(std::move(o.out)) // since lambda capture does not yet support
move
                                                                {
                                                                }
                                                                mutable iterator_type cursor;
                                                                iterator_type end;
                                                                mutable output_type out;
                                                        };

                                                        // creates a worker whose lifetime is the same as this subscription
                                                        auto coordinator = initial.coordination.create_coordinator(o.get_subscription());

                                                        iterate_state_type state(initial, o);

                                                        auto controller = coordinator.get_worker();

                                                        auto producer = [state](const rxsc::schedulable& self){
                                                                if (!state.out.is_subscribed()) {
                                                                        // terminate loop
                                                                        return;
                                                                }

                                                                if (state.cursor != state.end) {
                                                                        // send next value
                                                                        state.out.on_next(*state.cursor);
                                                                        ++state.cursor;
                                                                }

                                                                if (state.cursor == state.end) {
                                                                        state.out.on_completed();
                                                                        // o is unsubscribed
                                                                        return;
                                                                }

                                                                // tail recurse this same action to continue loop
                                                                self();
                                                        };
                                                        auto selectedProducer = on_exception(
                                                                [&](){return coordinator.act(producer); },
                                                                o);
                                                        if (selectedProducer.empty()) {
                                                                return;
                                                        }
                                                        controller.schedule(selectedProducer.get());

                                                }
                                        };

                        }

                        template<class Collection>
                        auto iterate(Collection c)
                                ->      observable<rxu::value_type_t<detail::iterate_traits<Collection>>, detail::iterate<Collection,
identity_one_worker>> {
                                return  observable<rxu::value_type_t<detail::iterate_traits<Collection>>, detail::iterate<Collection,
identity_one_worker>>(
                                        detail::iterate<Collection, identity_one_worker>(std::move(c), identity_immediate()));
                        }
                        template<class Collection, class Coordination>
                        auto iterate(Collection c, Coordination cn)
                                ->      observable<rxu::value_type_t<detail::iterate_traits<Collection>>, detail::iterate<Collection,
Coordination>> {
                                return  observable<rxu::value_type_t<detail::iterate_traits<Collection>>, detail::iterate<Collection,
Coordination>>(
                                        detail::iterate<Collection, Coordination>(std::move(c), std::move(cn)));
                        }

                        template<class T>
                        auto from()
                                -> decltype(iterate(std::array<T, 0>(), identity_immediate())) {
                                return      iterate(std::array<T, 0>(), identity_immediate());
                        }
                        template<class T, class Coordination>
                        auto from(Coordination cn)
                                -> typename std::enable_if<is_coordination<Coordination>::value,
                                decltype(iterate(std::array<T, 0>(), std::move(cn)))>::type {
                                return          iterate(std::array<T, 0>(), std::move(cn));
                        }
                        template<class Value0, class... ValueN>
                        auto from(Value0 v0, ValueN... vn)
                                -> typename std::enable_if<!is_coordination<Value0>::value,
                                decltype(iterate(*(std::array<Value0, sizeof...(ValueN)+1>*)nullptr, identity_immediate()))>::type {
```

```
                                        std::array<Value0, sizeof...(ValueN)+1> c{ { v0, vn... } };
                                        return iterate(std::move(c), identity_immediate());
                                }
                                template<class Coordination, class Value0, class... ValueN>
                                auto from(Coordination cn, Value0 v0, ValueN... vn)
                                        -> typename std::enable_if<is_coordination<Coordination>::value,
                                        decltype(iterate(*(std::array<Value0, sizeof...(ValueN)+1>*)nullptr, std::move(cn)))>::type {
                                        std::array<Value0, sizeof...(ValueN)+1> c{ { v0, vn... } };
                                        return iterate(std::move(c), std::move(cn));
                                }

                        }

}

#endif


#if !defined(RXCPP_SOURCES_RX_INTERVAL_HPP)
#define RXCPP_SOURCES_RX_INTERVAL_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace sources {

                namespace detail {

                        template<class Coordination>
                        struct interval : public source_base<long>
                        {
                                typedef interval<Coordination> this_type;

                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef typename coordination_type::coordinator_type coordinator_type;

                                struct interval_initial_type
                                {
                                        interval_initial_type(rxsc::scheduler::clock_type::time_point i,
rxsc::scheduler::clock_type::duration p, coordination_type cn)
                                        : initial(i)
                                        , period(p)
                                        , coordination(std::move(cn))
                                        {
                                        }
                                        rxsc::scheduler::clock_type::time_point initial;
                                        rxsc::scheduler::clock_type::duration period;
                                        coordination_type coordination;
                                };
                                interval_initial_type initial;

                                interval(rxsc::scheduler::clock_type::time_point i, rxsc::scheduler::clock_type::duration p,
coordination_type cn)
                                        : initial(i, p, std::move(cn))
                                {
                                }
                                template<class Subscriber>
                                void on_subscribe(Subscriber o) const {
                                        static_assert(is_subscriber<Subscriber>::value, "subscribe must be passed a subscriber");

                                        // creates a worker whose lifetime is the same as this subscription
                                        auto coordinator = initial.coordination.create_coordinator(o.get_subscription());

                                        auto controller = coordinator.get_worker();

                                        auto counter = std::make_shared<long>(0);

                                        auto producer = [o, counter](const rxsc::schedulable&) {
                                                // send next value
                                                o.on_next(++(*counter));
                                        };

                                        auto selectedProducer = on_exception(
                                                [&](){return coordinator.act(producer); },
                                                o);
                                        if (selectedProducer.empty()) {
                                                return;
                                        }

                                        controller.schedule_periodically(initial.initial, initial.period, selectedProducer.get());
                                }
```

```
                        };

                        template<class Duration, class Coordination>
                        struct defer_interval : public defer_observable<
                                rxu::all_true<
                                std::is_convertible<Duration, rxsc::scheduler::clock_type::duration>::value,
                                is_coordination<Coordination>::value>,
                                void,
                                interval, Coordination>
                        {
                        };

                }

                template<class Duration>
                auto interval(Duration period)
                        -> typename std::enable_if<
                        detail::defer_interval<Duration, identity_one_worker>::value,
                        typename    detail::defer_interval<Duration, identity_one_worker>::observable_type>::type {
                        return      detail::defer_interval<Duration, identity_one_worker>::make(identity_current_thread().now(),
period, identity_current_thread());
                }

                template<class Coordination>
                auto interval(rxsc::scheduler::clock_type::duration period, Coordination cn)
                        -> typename std::enable_if<
                        detail::defer_interval<rxsc::scheduler::clock_type::duration, Coordination>::value,
                        typename    detail::defer_interval<rxsc::scheduler::clock_type::duration, Coordination>::observable_type>::type
{
                        return      detail::defer_interval<rxsc::scheduler::clock_type::duration, Coordination>::make(cn.now(), period,
std::move(cn));
                }

                template<class Duration>
                auto interval(rxsc::scheduler::clock_type::time_point when, Duration period)
                        -> typename std::enable_if<
                        detail::defer_interval<Duration, identity_one_worker>::value,
                        typename    detail::defer_interval<Duration, identity_one_worker>::observable_type>::type {
                        return      detail::defer_interval<Duration, identity_one_worker>::make(when, period,
identity_current_thread());
                }

                template<class Coordination>
                auto interval(rxsc::scheduler::clock_type::time_point when, rxsc::scheduler::clock_type::duration period, Coordination cn)
                        -> typename std::enable_if<
                        detail::defer_interval<rxsc::scheduler::clock_type::duration, Coordination>::value,
                        typename    detail::defer_interval<rxsc::scheduler::clock_type::duration, Coordination>::observable_type>::type
{
                        return      detail::defer_interval<rxsc::scheduler::clock_type::duration, Coordination>::make(when, period,
std::move(cn));
                }

        }

}

#endif


#if !defined(RXCPP_SOURCES_RX_EMPTY_HPP)
#define RXCPP_SOURCES_RX_EMPTY_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace sources {

                template<class T>
                auto empty()
                        -> decltype(from<T>()) {
                        return      from<T>();
                }
                template<class T, class Coordination>
                auto empty(Coordination cn)
                        -> decltype(from<T>(std::move(cn))) {
                        return      from<T>(std::move(cn));
                }

        }

}
```

```
#endif


#if !defined(RXCPP_SOURCES_RX_DEFER_HPP)
#define RXCPP_SOURCES_RX_DEFER_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace sources {

		namespace detail {

			template<class ObservableFactory>
			struct defer_traits
			{
				typedef rxu::decay_t<ObservableFactory> observable_factory_type;
				typedef decltype((*(observable_factory_type*)nullptr)()) collection_type;
				typedef typename collection_type::value_type value_type;
			};

			template<class ObservableFactory>
			struct defer : public source_base<rxu::value_type_t<defer_traits<ObservableFactory>>>
			{
				typedef defer<ObservableFactory> this_type;
				typedef defer_traits<ObservableFactory> traits;

				typedef typename traits::observable_factory_type observable_factory_type;
				typedef typename traits::collection_type collection_type;

				observable_factory_type observable_factory;

				defer(observable_factory_type of)
					: observable_factory(std::move(of))
				{
				}
				template<class Subscriber>
				void on_subscribe(Subscriber o) const {

					auto selectedCollection = on_exception(
						[this](){return this->observable_factory(); },
						o);
					if (selectedCollection.empty()) {
						return;
					}

					selectedCollection->subscribe(o);
				}
			};

		}

		template<class ObservableFactory>
		auto defer(ObservableFactory of)
			->	observable<rxu::value_type_t<detail::defer_traits<ObservableFactory>>,
detail::defer<ObservableFactory>> {
			return  observable<rxu::value_type_t<detail::defer_traits<ObservableFactory>>,
detail::defer<ObservableFactory>>(
				detail::defer<ObservableFactory>(std::move(of)));
		}

	}

}

#endif


#if !defined(RXCPP_SOURCES_RX_NEVER_HPP)
#define RXCPP_SOURCES_RX_NEVER_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace sources {

		namespace detail {

			template<class T>
```

```
                struct never : public source_base<T>
                {
                        template<class Subscriber>
                        void on_subscribe(Subscriber) const {
                        }
                };

        }

        template<class T>
        auto never()
                ->      observable<T, detail::never<T>> {
                return  observable<T, detail::never<T>>(detail::never<T>());
        }

    }

}

#endif


#if !defined(RXCPP_SOURCES_RX_ERROR_HPP)
#define RXCPP_SOURCES_RX_ERROR_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

    namespace sources {

        namespace detail {

            template<class T, class Coordination>
            struct error : public source_base<T>
            {
                typedef error<T, Coordination> this_type;

                typedef rxu::decay_t<Coordination> coordination_type;

                typedef typename coordination_type::coordinator_type coordinator_type;

                struct error_initial_type
                {
                    error_initial_type(std::exception_ptr e, coordination_type cn)
                    : exception(e)
                    , coordination(std::move(cn))
                    {
                    }
                    std::exception_ptr exception;
                    coordination_type coordination;
                };
                error_initial_type initial;

                error(std::exception_ptr e, coordination_type cn)
                    : initial(e, std::move(cn))
                {
                }

                template<class Subscriber>
                void on_subscribe(Subscriber o) const {

                    // creates a worker whose lifetime is the same as this subscription
                    auto coordinator = initial.coordination.create_coordinator(o.get_subscription());
                    auto controller = coordinator.get_worker();
                    auto exception = initial.exception;

                    auto producer = [=](const rxsc::schedulable&){
                        auto& dest = o;
                        if (!dest.is_subscribed()) {
                            // terminate loop
                            return;
                        }

                        dest.on_error(exception);
                        // o is unsubscribed
                    };
                    auto selectedProducer = on_exception(
                        [&](){return coordinator.act(producer); },
                        o);
                    if (selectedProducer.empty()) {
                        return;
```

```
                                        }
                                        controller.schedule(selectedProducer.get());
                                }
                        };

                        struct throw_ptr_tag{};
                        struct throw_instance_tag{};

                        template <class T, class Coordination>
                        auto make_error(throw_ptr_tag&&, std::exception_ptr exception, Coordination cn)
                                ->      observable<T, error<T, Coordination>> {
                                return  observable<T, error<T, Coordination>>(error<T, Coordination>(std::move(exception),
std::move(cn)));
                        }

                        template <class T, class E, class Coordination>
                        auto make_error(throw_instance_tag&&, E e, Coordination cn)
                                ->      observable<T, error<T, Coordination>> {
                                std::exception_ptr exception;
                                try { throw e; }
                                catch (...) { exception = std::current_exception(); }
                                return  observable<T, error<T, Coordination>>(error<T, Coordination>(std::move(exception),
std::move(cn)));
                        }

                }

                template<class T, class E>
                auto error(E e)
                        -> decltype(detail::make_error<T>(typename std::conditional<std::is_same<std::exception_ptr,
rxu::decay_t<E>>::value, detail::throw_ptr_tag, detail::throw_instance_tag>::type(), std::move(e), identity_immediate())) {
                        return      detail::make_error<T>(typename std::conditional<std::is_same<std::exception_ptr,
rxu::decay_t<E>>::value, detail::throw_ptr_tag, detail::throw_instance_tag>::type(), std::move(e), identity_immediate());
                }
                template<class T, class E, class Coordination>
                auto error(E e, Coordination cn)
                        -> decltype(detail::make_error<T>(typename std::conditional<std::is_same<std::exception_ptr,
rxu::decay_t<E>>::value, detail::throw_ptr_tag, detail::throw_instance_tag>::type(), std::move(e), std::move(cn))) {
                        return      detail::make_error<T>(typename std::conditional<std::is_same<std::exception_ptr,
rxu::decay_t<E>>::value, detail::throw_ptr_tag, detail::throw_instance_tag>::type(), std::move(e), std::move(cn));
                }

        }

}

#endif


#if !defined(RXCPP_SOURCES_RX_SCOPE_HPP)
#define RXCPP_SOURCES_RX_SCOPE_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace sources {

                namespace detail {

                        template<class ResourceFactory, class ObservableFactory>
                        struct scope_traits
                        {
                                typedef rxu::decay_t<ResourceFactory> resource_factory_type;
                                typedef rxu::decay_t<ObservableFactory> observable_factory_type;
                                typedef decltype((*(resource_factory_type*)nullptr)()) resource_type;
                                typedef decltype((*(observable_factory_type*)nullptr)(resource_type())) collection_type;
                                typedef typename collection_type::value_type value_type;

                                static_assert(is_subscription<resource_type>::value, "ResourceFactory must return a subscription");
                        };

                        template<class ResourceFactory, class ObservableFactory>
                        struct scope : public source_base<rxu::value_type_t<scope_traits<ResourceFactory, ObservableFactory>>>
                        {
                                typedef scope_traits<ResourceFactory, ObservableFactory> traits;
                                typedef typename traits::resource_factory_type resource_factory_type;
                                typedef typename traits::observable_factory_type observable_factory_type;
                                typedef typename traits::resource_type resource_type;
                                typedef typename traits::value_type value_type;

                                struct values
```

```
                                        {
                                                values(resource_factory_type rf, observable_factory_type of)
                                                : resource_factory(std::move(rf))
                                                , observable_factory(std::move(of))
                                                {
                                                }
                                                resource_factory_type resource_factory;
                                                observable_factory_type observable_factory;
                                        };
                                        values initial;


                                        scope(resource_factory_type rf, observable_factory_type of)
                                                : initial(std::move(rf), std::move(of))
                                        {
                                        }

                                        template<class Subscriber>
                                        void on_subscribe(Subscriber o) const {

                                                struct state_type
                                                        : public std::enable_shared_from_this<state_type>
                                                        , public values
                                                {
                                                        state_type(values i, Subscriber o)
                                                        : values(i)
                                                        , out(std::move(o))
                                                        {
                                                        }
                                                        Subscriber out;
                                                        rxu::detail::maybe<resource_type> resource;
                                                };

                                                auto state = std::make_shared<state_type>(state_type(initial, std::move(o)));

                                                state->resource = on_exception(
                                                        [&](){return state->resource_factory(); },
                                                        state->out);
                                                if (state->resource.empty()) {
                                                        return;
                                                }
                                                state->out.add(state->resource->get_subscription());

                                                auto selectedCollection = on_exception(
                                                        [state](){return state->observable_factory(state->resource.get()); },
                                                        state->out);
                                                if (selectedCollection.empty()) {
                                                        return;
                                                }

                                                selectedCollection->subscribe(state->out);
                                        }
                                };

                }

                template<class ResourceFactory, class ObservableFactory>
                auto scope(ResourceFactory rf, ObservableFactory of)
                        ->      observable<rxu::value_type_t<detail::scope_traits<ResourceFactory, ObservableFactory>>,
detail::scope<ResourceFactory, ObservableFactory>> {
                        return  observable<rxu::value_type_t<detail::scope_traits<ResourceFactory, ObservableFactory>>,
detail::scope<ResourceFactory, ObservableFactory>>(
                                detail::scope<ResourceFactory, ObservableFactory>(std::move(rf), std::move(of)));
                }

        }

}

#endif


#if !defined(RXCPP_SOURCES_RX_TIMER_HPP)
#define RXCPP_SOURCES_RX_TIMER_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace sources {

                namespace detail {
```

```
template<class Coordination>
struct timer : public source_base<long>
{
	typedef timer<Coordination> this_type;

	typedef rxu::decay_t<Coordination> coordination_type;
	typedef typename coordination_type::coordinator_type coordinator_type;

	struct timer_initial_type
	{
		timer_initial_type(rxsc::scheduler::clock_type::time_point t, coordination_type cn)
		: when(t)
		, coordination(std::move(cn))
		{
		}
		rxsc::scheduler::clock_type::time_point when;
		coordination_type coordination;
	};
	timer_initial_type initial;

	timer(rxsc::scheduler::clock_type::time_point t, coordination_type cn)
		: initial(t, std::move(cn))
	{
	}
	timer(rxsc::scheduler::clock_type::duration p, coordination_type cn)
		: initial(rxsc::scheduler::clock_type::time_point(), std::move(cn))
	{
		initial.when = initial.coordination.now() + p;
	}
	template<class Subscriber>
	void on_subscribe(Subscriber o) const {
		static_assert(is_subscriber<Subscriber>::value, "subscribe must be passed a subscriber");

		// creates a worker whose lifetime is the same as this subscription
		auto coordinator = initial.coordination.create_coordinator(o.get_subscription());
		auto controller = coordinator.get_worker();

		auto producer = [o](const rxsc::schedulable&) {
			// send the value and complete
			o.on_next(1L);
			o.on_completed();
		};

		auto selectedProducer = on_exception(
			[&](){return coordinator.act(producer); },
			o);
		if (selectedProducer.empty()) {
			return;
		}

		controller.schedule(initial.when, selectedProducer.get());
	}
};

template<class TimePointOrDuration, class Coordination>
struct defer_timer : public defer_observable<
	rxu::all_true<
	std::is_convertible<TimePointOrDuration, rxsc::scheduler::clock_type::time_point>::value ||
	std::is_convertible<TimePointOrDuration, rxsc::scheduler::clock_type::duration>::value,
	is_coordination<Coordination>::value>,
	void,
	timer, Coordination>
{
};

}

template<class TimePointOrDuration>
auto timer(TimePointOrDuration when)
	-> typename std::enable_if<
	detail::defer_timer<TimePointOrDuration, identity_one_worker>::value,
	typename detail::defer_timer<TimePointOrDuration, identity_one_worker>::observable_type>::type {
	return detail::defer_timer<TimePointOrDuration, identity_one_worker>::make(when,
identity_current_thread());
}

template<class TimePointOrDuration, class Coordination>
auto timer(TimePointOrDuration when, Coordination cn)
	-> typename std::enable_if<
	detail::defer_timer<TimePointOrDuration, Coordination>::value,
	typename detail::defer_timer<TimePointOrDuration, Coordination>::observable_type>::type {
```

```
                                return          detail::defer_timer<TimePointOrDuration, Coordination>::make(when, std::move(cn));
                        }

                }

}

#endif


#endif


#if !defined(RXCPP_RX_SCHEDULER_SUBJECTS_HPP)
#define RXCPP_RX_SCHEDULER_SUBJECTS_HPP

//_include "rx-includes.hpp"

namespace rxcpp {

        namespace subjects {

        }
        namespace rxsub = subjects;

}

#if !defined(RXCPP_RX_SUBJECT_HPP)
#define RXCPP_RX_SUBJECT_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace subjects {

                namespace detail {

                        template<class T>
                        class multicast_observer
                        {
                                typedef subscriber<T> observer_type;
                                typedef std::vector<observer_type> list_type;

                                struct mode
                                {
                                        enum type {
                                                Invalid = 0,
                                                Casting,
                                                Disposed,
                                                Completed,
                                                Errored
                                        };
                                };

                                struct state_type
                                        : public std::enable_shared_from_this<state_type>
                                {
                                        explicit state_type(composite_subscription cs)
                                        : generation(0)
                                        , current(mode::Casting)
                                        , lifetime(cs)
                                        {
                                        }
                                        std::atomic<int> generation;
                                        std::mutex lock;
                                        typename mode::type current;
                                        std::exception_ptr error;
                                        composite_subscription lifetime;
                                };

                                struct completer_type
                                        : public std::enable_shared_from_this<completer_type>
                                {
                                        ~completer_type()
                                        {
                                        }
                                        completer_type(std::shared_ptr<state_type> s, const std::shared_ptr<completer_type>&
old, observer_type o)
                                                : state(s)
                                        {
                                                retain(old);
```

```
                                                observers.push_back(o);
                                        }
                                        completer_type(std::shared_ptr<state_type> s, const std::shared_ptr<completer_type>&
old)
                                                : state(s)
                                        {
                                                retain(old);
                                        }
                                        void retain(const std::shared_ptr<completer_type>& old) {
                                                if (old) {
                                                        observers.reserve(old->observers.size() + 1);
                                                        std::copy_if(
                                                                old->observers.begin(), old->observers.end(),
                                                                std::inserter(observers, observers.end()),
                                                                [](const observer_type& o){
                                                                return o.is_subscribed();
                                                        });
                                                }
                                        }
                                        std::shared_ptr<state_type> state;
                                        list_type observers;
                                };

                                // this type prevents a circular ref between state and completer
                                struct binder_type
                                        : public std::enable_shared_from_this<binder_type>
                                {
                                        explicit binder_type(composite_subscription cs)
                                        : state(std::make_shared<state_type>(cs))
                                        , id(trace_id::make_next_id_subscriber())
                                        , current_generation(0)
                                        {
                                        }

                                        std::shared_ptr<state_type> state;

                                        trace_id id;

                                        // used to avoid taking lock in on_next
                                        mutable int current_generation;
                                        mutable std::shared_ptr<completer_type> current_completer;

                                        // must only be accessed under state->lock
                                        mutable std::shared_ptr<completer_type> completer;
                                };

                                std::shared_ptr<binder_type> b;

                        public:
                                typedef subscriber<T, observer<T, detail::multicast_observer<T>>> input_subscriber_type;

                                explicit multicast_observer(composite_subscription cs)
                                        : b(std::make_shared<binder_type>(cs))
                                {
                                        std::weak_ptr<binder_type> binder = b;
                                        b->state->lifetime.add([binder](){
                                                auto b = binder.lock();
                                                if (b && b->state->current == mode::Casting){
                                                        b->state->current = mode::Disposed;
                                                        b->current_completer.reset();
                                                        b->completer.reset();
                                                        ++b->state->generation;
                                                }
                                        });
                                }
                                trace_id get_id() const {
                                        return b->id;
                                }
                                composite_subscription get_subscription() const {
                                        return b->state->lifetime;
                                }
                                input_subscriber_type get_subscriber() const {
                                        return make_subscriber<T>(get_id(), get_subscription(), observer<T,
detail::multicast_observer<T>>(*this));
                                }
                                bool has_observers() const {
                                        std::unique_lock<std::mutex> guard(b->state->lock);
                                        return b->current_completer && !b->current_completer->observers.empty();
                                }
                                template<class SubscriberFrom>
                                void add(const SubscriberFrom& sf, observer_type o) const {
                                        trace_activity().connect(sf, o);
```

```
                                        std::unique_lock<std::mutex> guard(b->state->lock);
                                        switch (b->state->current) {
                                        case mode::Casting:
                                        {
                                                                                                if (o.is_subscribed()) {

std::weak_ptr<binder_type> binder = b;
                                                                                                        o.add([=](){
                                                                                                                auto b =
binder.lock();
                                                                                                                if (b) {

std::unique_lock<std::mutex> guard(b->state->lock);

b->completer = std::make_shared<completer_type>(b->state, b->completer);

++b->state->generation;
                                                                                                                }
                                                                                                        });
                                                                                                        b->completer =
std::make_shared<completer_type>(b->state, b->completer, o);
                                                                                                        ++b->state-
>generation;
                                                                                                }
                                        }
                                                break;
                                        case mode::Completed:
                                        {
                                                                                                        guard.unlock();
                                                                                                        o.on_completed();
                                                                                                        return;
                                        }
                                                break;
                                        case mode::Errored:
                                        {
                                                                                                auto e = b->state->error;
                                                                                                guard.unlock();
                                                                                                o.on_error(e);
                                                                                                return;
                                        }
                                                break;
                                        case mode::Disposed:
                                        {
                                                                                                 guard.unlock();
                                                                                                 o.unsubscribe();
                                                                                                 return;
                                        }
                                                break;
                                        default:
                                                std::terminate();
                                        }
                                }
                                template<class V>
                                void on_next(V v) const {
                                        if (b->current_generation != b->state->generation) {
                                                std::unique_lock<std::mutex> guard(b->state->lock);
                                                b->current_generation = b->state->generation;
                                                b->current_completer = b->completer;
                                        }

                                        auto current_completer = b->current_completer;
                                        if (!current_completer || current_completer->observers.empty()) {
                                                return;
                                        }
                                        for (auto& o : current_completer->observers) {
                                                if (o.is_subscribed()) {
                                                        o.on_next(v);
                                                }
                                        }
                                }
                                void on_error(std::exception_ptr e) const {
                                        std::unique_lock<std::mutex> guard(b->state->lock);
                                        if (b->state->current == mode::Casting) {
                                                b->state->error = e;
                                                b->state->current = mode::Errored;
                                                auto s = b->state->lifetime;
                                                auto c = std::move(b->completer);
                                                b->current_completer.reset();
                                                ++b->state->generation;
                                                guard.unlock();
                                                if (c) {
                                                        for (auto& o : c->observers) {
```

```
                                                                        if (o.is_subscribed()) {
                                                                                o.on_error(e);
                                                                        }
                                                                }
                                                        }
                                                        s.unsubscribe();
                                                }
                                        }
                                        void on_completed() const {
                                                std::unique_lock<std::mutex> guard(b->state->lock);
                                                if (b->state->current == mode::Casting) {
                                                        b->state->current = mode::Completed;
                                                        auto s = b->state->lifetime;
                                                        auto c = std::move(b->completer);
                                                        b->current_completer.reset();
                                                        ++b->state->generation;
                                                        guard.unlock();
                                                        if (c) {
                                                                for (auto& o : c->observers) {
                                                                        if (o.is_subscribed()) {
                                                                                o.on_completed();
                                                                        }
                                                                }
                                                        }
                                                        s.unsubscribe();
                                                }
                                        }
                                };


                }

                template<class T>
                class subject
                {
                        detail::multicast_observer<T> s;

                public:
                        typedef subscriber<T, observer<T, detail::multicast_observer<T>>> subscriber_type;
                        typedef observable<T> observable_type;
                        subject()
                                : s(composite_subscription())
                        {
                        }
                        explicit subject(composite_subscription cs)
                                : s(cs)
                        {
                        }

                        bool has_observers() const {
                                return s.has_observers();
                        }

                        subscriber_type get_subscriber() const {
                                return s.get_subscriber();
                        }

                        observable<T> get_observable() const {
                                auto keepAlive = s;
                                return make_observable_dynamic<T>([=](subscriber<T> o){
                                        keepAlive.add(keepAlive.get_subscriber(), std::move(o));
                                });
                        }
                };

        }

}

#endif


#if !defined(RXCPP_RX_BEHAVIOR_HPP)
#define RXCPP_RX_BEHAVIOR_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace subjects {

                namespace detail {
```

```
template<class T>
class behavior_observer : public detail::multicast_observer<T>
{
	typedef behavior_observer<T> this_type;
	typedef detail::multicast_observer<T> base_type;

	class behavior_observer_state : public std::enable_shared_from_this<behavior_observer_state>
	{
		mutable std::mutex lock;
		mutable T value;

	public:
		behavior_observer_state(T first)
			: value(first)
		{
		}

		void reset(T v) const {
			std::unique_lock<std::mutex> guard(lock);
			value = std::move(v);
		}
		T get() const {
			std::unique_lock<std::mutex> guard(lock);
			return value;
		}
	};

	std::shared_ptr<behavior_observer_state> state;

public:
	behavior_observer(T f, composite_subscription l)
		: base_type(l)
		, state(std::make_shared<behavior_observer_state>(std::move(f)))
	{
	}

	subscriber<T> get_subscriber() const {
		return make_subscriber<T>(this->get_id(), this->get_subscription(), observer<T,
detail::behavior_observer<T>>(*this)).as_dynamic();
	}

	T get_value() const {
		return state->get();
	}

	template<class V>
	void on_next(V v) const {
		state->reset(v);
		base_type::on_next(std::move(v));
	}
};

}

template<class T>
class behavior
{
	detail::behavior_observer<T> s;

public:
	explicit behavior(T f, composite_subscription cs = composite_subscription())
		: s(std::move(f), cs)
	{
	}

	bool has_observers() const {
		return s.has_observers();
	}

	T get_value() const {
		return s.get_value();
	}

	subscriber<T> get_subscriber() const {
		return s.get_subscriber();
	}

	observable<T> get_observable() const {
		auto keepAlive = s;
		return make_observable_dynamic<T>([=](subscriber<T> o){
			if (keepAlive.get_subscription().is_subscribed()) {
```

```
                                                o.on_next(get_value());
                                        }
                                        keepAlive.add(s.get_subscriber(), std::move(o));
                                });
                        }
                };

        }

}

#endif


#if !defined(RXCPP_RX_REPLAYSUBJECT_HPP)
#define RXCPP_RX_REPLAYSUBJECT_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace subjects {

                namespace detail {

                        template<class Coordination>
                        struct replay_traits
                        {
                                typedef rxu::maybe<std::size_t> count_type;
                                typedef rxu::maybe<rxsc::scheduler::clock_type::duration> period_type;
                                typedef rxsc::scheduler::clock_type::time_point time_point_type;
                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef typename coordination_type::coordinator_type coordinator_type;
                        };

                        template<class T, class Coordination>
                        class replay_observer : public detail::multicast_observer<T>
                        {
                                typedef replay_observer<T, Coordination> this_type;
                                typedef detail::multicast_observer<T> base_type;

                                typedef replay_traits<Coordination> traits;
                                typedef typename traits::count_type count_type;
                                typedef typename traits::period_type period_type;
                                typedef typename traits::time_point_type time_point_type;
                                typedef typename traits::coordination_type coordination_type;
                                typedef typename traits::coordinator_type coordinator_type;

                                class replay_observer_state : public std::enable_shared_from_this<replay_observer_state>
                                {
                                        mutable std::mutex lock;
                                        mutable std::list<T> values;
                                        mutable std::list<time_point_type> time_points;
                                        mutable count_type count;
                                        mutable period_type period;
                                public:
                                        mutable coordination_type coordination;
                                        mutable coordinator_type coordinator;

                                private:
                                        void remove_oldest() const {
                                                values.pop_front();
                                                if (!period.empty()) {
                                                        time_points.pop_front();
                                                }
                                        }

                                public:
                                        explicit replay_observer_state(count_type _count, period_type _period, coordination_type
_coordination, coordinator_type _coordinator)
                                                : count(_count)
                                                , period(_period)
                                                , coordination(std::move(_coordination))
                                                , coordinator(std::move(_coordinator))
                                        {
                                        }

                                        void add(T v) const {
                                                std::unique_lock<std::mutex> guard(lock);
                                                if (!count.empty()) {
                                                        if (values.size() == count.get())
                                                                remove_oldest();
```

```
                                                }

                                                if (!period.empty()) {
                                                        auto now = coordination.now();
                                                        while (!time_points.empty() && (now - time_points.front() >
period.get()))
                                                                remove_oldest();
                                                        time_points.push_back(now);
                                                }

                                                values.push_back(std::move(v));
                                        }
                                        std::list<T> get() const {
                                                std::unique_lock<std::mutex> guard(lock);
                                                return values;
                                        }
                                };

                                std::shared_ptr<replay_observer_state> state;

                        public:
                                replay_observer(count_type count, period_type period, coordination_type coordination,
composite_subscription cs)
                                        : base_type(cs)
                                {
                                        auto coordinator = coordination.create_coordinator(cs);
                                        state = std::make_shared<replay_observer_state>(std::move(count), std::move(period),
std::move(coordination), std::move(coordinator));
                                }

                                subscriber<T> get_subscriber() const {
                                        return make_subscriber<T>(this->get_id(), this->get_subscription(), observer<T,
detail::replay_observer<T, Coordination>>(*this)).as_dynamic();
                                }

                                std::list<T> get_values() const {
                                        return state->get();
                                }

                                coordinator_type& get_coordinator() const {
                                        return state->coordinator;
                                }

                                template<class V>
                                void on_next(V v) const {
                                        state->add(v);
                                        base_type::on_next(std::move(v));
                                }
                        };

                }

                template<class T, class Coordination>
                class replay
                {
                        typedef detail::replay_traits<Coordination> traits;
                        typedef typename traits::count_type count_type;
                        typedef typename traits::period_type period_type;
                        typedef typename traits::time_point_type time_point_type;

                        detail::replay_observer<T, Coordination> s;

                public:
                        explicit replay(Coordination cn, composite_subscription cs = composite_subscription())
                                : s(count_type(), period_type(), cn, cs)
                        {
                        }

                        replay(std::size_t count, Coordination cn, composite_subscription cs = composite_subscription())
                                : s(count_type(std::move(count)), period_type(), cn, cs)
                        {
                        }

                        replay(rxsc::scheduler::clock_type::duration period, Coordination cn, composite_subscription cs =
composite_subscription())
                                : s(count_type(), period_type(period), cn, cs)
                        {
                        }

                        replay(std::size_t count, rxsc::scheduler::clock_type::duration period, Coordination cn, composite_subscription
cs = composite_subscription())
                                : s(count_type(count), period_type(period), cn, cs)
```

```
                                {
                                }

                                bool has_observers() const {
                                        return s.has_observers();
                                }

                                std::list<T> get_values() const {
                                        return s.get_values();
                                }

                                subscriber<T> get_subscriber() const {
                                        return s.get_subscriber();
                                }

                                observable<T> get_observable() const {
                                        auto keepAlive = s;
                                        auto observable = make_observable_dynamic<T>([=](subscriber<T> o){
                                                if (keepAlive.get_subscription().is_subscribed()) {
                                                        for (auto&& value : get_values())
                                                                o.on_next(value);
                                                }
                                                keepAlive.add(keepAlive.get_subscriber(), std::move(o));
                                        });
                                        return s.get_coordinator().in(observable);
                                }
                        };

                }

}

#endif


#if !defined(RXCPP_RX_SYNCHRONIZE_HPP)
#define RXCPP_RX_SYNCHRONIZE_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace subjects {

                namespace detail {

                        template<class T, class Coordination>
                        class synchronize_observer : public detail::multicast_observer<T>
                        {
                                typedef synchronize_observer<T, Coordination> this_type;
                                typedef detail::multicast_observer<T> base_type;

                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef typename coordination_type::coordinator_type coordinator_type;
                                typedef typename coordinator_type::template get<subscriber<T>>::type output_type;

                                struct synchronize_observer_state : public
std::enable_shared_from_this<synchronize_observer_state>
                                {
                                        typedef rxn::notification<T> notification_type;
                                        typedef typename notification_type::type base_notification_type;
                                        typedef std::deque<base_notification_type> queue_type;

                                        struct mode
                                        {
                                                enum type {
                                                        Invalid = 0,
                                                        Processing,
                                                        Empty,
                                                        Disposed
                                                };
                                        };

                                        mutable std::mutex lock;
                                        mutable std::condition_variable wake;
                                        mutable queue_type fill_queue;
                                        composite_subscription lifetime;
                                        mutable typename mode::type current;
                                        coordinator_type coordinator;
                                        output_type destination;

                                        void ensure_processing(std::unique_lock<std::mutex>& guard) const {
```

```
                                        if (!guard.owns_lock()) {
                                                std::terminate();
                                        }
                                        if (current == mode::Empty) {
                                                current = mode::Processing;
                                                auto keepAlive = this->shared_from_this();

                                                auto drain_queue = [keepAlive, this](const rxsc::schedulable& self){
                                                        try {
                                                                std::unique_lock<std::mutex> guard(lock);
                                                                if (!destination.is_subscribed()) {
                                                                        current = mode::Disposed;
                                                                        fill_queue.clear();
                                                                        guard.unlock();
                                                                        lifetime.unsubscribe();
                                                                        return;
                                                                }
                                                                if (fill_queue.empty()) {
                                                                        current = mode::Empty;
                                                                        return;
                                                                }
                                                                auto notification =
std::move(fill_queue.front());
                                                                fill_queue.pop_front();
                                                                guard.unlock();
                                                                notification->accept(destination);
                                                                self();
                                                        }
                                                        catch (...) {
                                                                destination.on_error(std::current_exception());
                                                                std::unique_lock<std::mutex> guard(lock);
                                                                current = mode::Empty;
                                                        }
                                                };

                                                auto selectedDrain = on_exception(
                                                        [&](){return coordinator.act(drain_queue); },
                                                        destination);
                                                if (selectedDrain.empty()) {
                                                        return;
                                                }

                                                auto processor = coordinator.get_worker();
                                                processor.schedule(lifetime, selectedDrain.get());
                                        }
                                }

                                synchronize_observer_state(coordinator_type coor, composite_subscription cs, output_type
scbr)
                                        : lifetime(std::move(cs))
                                        , current(mode::Empty)
                                        , coordinator(std::move(coor))
                                        , destination(std::move(scbr))
                                {
                                }

                                template<class V>
                                void on_next(V v) const {
                                        if (lifetime.is_subscribed()) {
                                                std::unique_lock<std::mutex> guard(lock);
                                                fill_queue.push_back(notification_type::on_next(std::move(v)));
                                                ensure_processing(guard);
                                        }
                                        wake.notify_one();
                                }
                                void on_error(std::exception_ptr e) const {
                                        if (lifetime.is_subscribed()) {
                                                std::unique_lock<std::mutex> guard(lock);
                                                fill_queue.push_back(notification_type::on_error(e));
                                                ensure_processing(guard);
                                        }
                                        wake.notify_one();
                                }
                                void on_completed() const {
                                        if (lifetime.is_subscribed()) {
                                                std::unique_lock<std::mutex> guard(lock);
                                                fill_queue.push_back(notification_type::on_completed());
                                                ensure_processing(guard);
                                        }
                                        wake.notify_one();
                                }
                        };
```

```
                                        std::shared_ptr<synchronize_observer_state> state;

                                public:
                                        synchronize_observer(coordination_type cn, composite_subscription dl, composite_subscription il)
                                                : base_type(dl)
                                        {
                                                auto o = make_subscriber<T>(dl,
make_observer_dynamic<T>(*static_cast<base_type*>(this)));

                                                // creates a worker whose lifetime is the same as the destination subscription
                                                auto coordinator = cn.create_coordinator(dl);

                                                state = std::make_shared<synchronize_observer_state>(std::move(coordinator),
std::move(il), std::move(o));
                                        }

                                        subscriber<T> get_subscriber() const {
                                                return make_subscriber<T>(this->get_id(), state->lifetime, observer<T,
detail::synchronize_observer<T, Coordination>>(*this)).as_dynamic();
                                        }

                                        template<class V>
                                        void on_next(V v) const {
                                                state->on_next(std::move(v));
                                        }
                                        void on_error(std::exception_ptr e) const {
                                                state->on_error(e);
                                        }
                                        void on_completed() const {
                                                state->on_completed();
                                        }
                                };

                        }

                        template<class T, class Coordination>
                        class synchronize
                        {
                                detail::synchronize_observer<T, Coordination> s;

                        public:
                                explicit synchronize(Coordination cn, composite_subscription cs = composite_subscription())
                                        : s(std::move(cn), std::move(cs), composite_subscription())
                                {
                                }

                                bool has_observers() const {
                                        return s.has_observers();
                                }

                                subscriber<T> get_subscriber() const {
                                        return s.get_subscriber();
                                }

                                observable<T> get_observable() const {
                                        auto keepAlive = s;
                                        return make_observable_dynamic<T>([=](subscriber<T> o){
                                                keepAlive.add(keepAlive.get_subscriber(), std::move(o));
                                        });
                                }
                        };

                }

                class synchronize_in_one_worker : public coordination_base
                {
                        rxsc::scheduler factory;

                        class input_type
                        {
                                rxsc::worker controller;
                                rxsc::scheduler factory;
                                identity_one_worker coordination;
                        public:
                                explicit input_type(rxsc::worker w)
                                        : controller(w)
                                        , factory(rxsc::make_same_worker(w))
                                        , coordination(factory)
                                {
                                }
                                inline rxsc::worker get_worker() const {
```

```
                                        return controller;
                                }
                                inline rxsc::scheduler get_scheduler() const {
                                        return factory;
                                }
                                inline rxsc::scheduler::clock_type::time_point now() const {
                                        return factory.now();
                                }
                                template<class Observable>
                                auto in(Observable o) const
                                        -> decltype(o.publish_synchronized(coordination).ref_count()) {
                                        return      o.publish_synchronized(coordination).ref_count();
                                }
                                template<class Subscriber>
                                auto out(Subscriber s) const
                                        -> Subscriber {
                                        return s;
                                }
                                template<class F>
                                auto act(F f) const
                                        -> F {
                                        return f;
                                }
                        };

                public:

                        explicit synchronize_in_one_worker(rxsc::scheduler sc) : factory(sc) {}

                        typedef coordinator<input_type> coordinator_type;

                        inline rxsc::scheduler::clock_type::time_point now() const {
                                return factory.now();
                        }

                        inline coordinator_type create_coordinator(composite_subscription cs = composite_subscription()) const {
                                auto w = factory.create_worker(std::move(cs));
                                return coordinator_type(input_type(std::move(w)));
                        }
                };

                inline synchronize_in_one_worker synchronize_event_loop() {
                        static synchronize_in_one_worker r(rxsc::make_event_loop());
                        return r;
                }

                inline synchronize_in_one_worker synchronize_new_thread() {
                        static synchronize_in_one_worker r(rxsc::make_new_thread());
                        return r;
                }

}

#endif


#endif


#if !defined(RXCPP_RX_OPERATORS_HPP)
#define RXCPP_RX_OPERATORS_HPP

//_include "rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                struct tag_operator {};
                template<class T>
                struct operator_base
                {
                        typedef T value_type;
                        typedef tag_operator operator_tag;
                };

                namespace detail {

                        template<class T, class = rxu::types_checked>
                        struct is_operator : std::false_type
                        {
                        };
```

```
                template<class T>
                struct is_operator<T, rxu::types_checked_t<typename T::operator_tag>>
                        : std::is_convertible<typename T::operator_tag*, tag_operator*>
                {
                };

        }

        template<class T, class Decayed = rxu::decay_t<T>>
        struct is_operator : detail::is_operator<Decayed>
        {
        };


}
namespace rxo = operators;

template<class Tag>
struct member_overload
{
        template<class... AN>
        static auto member(AN&&...) ->
                typename Tag::template include_header<std::false_type> {
                return  typename Tag::template include_header<std::false_type>();
        }
};

template<class T, class... AN>
struct delayed_type{ using value_type = T; static T value(AN**...) { return T{}; } };

template<class T, class... AN>
using delayed_type_t = rxu::value_type_t<delayed_type<T, AN...>>;

template<class Tag, class... AN, class Overload = member_overload<rxu::decay_t<Tag>>>
auto observable_member(Tag, AN&&... an) ->
        decltype(Overload::member(std::forward<AN>(an)...)) {
        return   Overload::member(std::forward<AN>(an)...);
}

template<class Tag, class... AN>
class operator_factory
{
        using this_type = operator_factory<Tag, AN...>;
        using tag_type = rxu::decay_t<Tag>;
        using tuple_type = std::tuple<rxu::decay_t<AN>...>;

        tuple_type an;

public:
        operator_factory(tuple_type an)
                : an(std::move(an))
        {
        }

        template<class... ZN>
        auto operator()(tag_type t, ZN&&... zn) const
                -> decltype(observable_member(t, std::forward<ZN>(zn)...)) {
                return      observable_member(t, std::forward<ZN>(zn)...);
        }

        template<class Observable>
        auto operator()(Observable source) const
                -> decltype(rxu::apply(std::tuple_cat(std::make_tuple(tag_type{}, source), (*(tuple_type*)nullptr)),
(*(this_type*)nullptr))) {
                return      rxu::apply(std::tuple_cat(std::make_tuple(tag_type{}, source), an), *this);
        }
};

}


#if !defined(RXCPP_OPERATORS_RX_AMB_HPP)
#define RXCPP_OPERATORS_RX_AMB_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {
```

```
template<class T, class Observable, class Coordination>
struct amb
	: public operator_base<rxu::value_type_t<T>>
{
	//static_assert(is_observable<Observable>::value, "amb requires an observable");
	//static_assert(is_observable<T>::value, "amb requires an observable that contains observables");

	typedef amb<T, Observable, Coordination> this_type;

	typedef rxu::decay_t<T> source_value_type;
	typedef rxu::decay_t<Observable> source_type;

	typedef typename source_type::source_operator_type source_operator_type;
	typedef typename source_value_type::value_type value_type;

	typedef rxu::decay_t<Coordination> coordination_type;
	typedef typename coordination_type::coordinator_type coordinator_type;

	struct values
	{
		values(source_operator_type o, coordination_type sf)
		: source_operator(std::move(o))
		, coordination(std::move(sf))
		{
		}
		source_operator_type source_operator;
		coordination_type coordination;
	};
	values initial;

	amb(const source_type& o, coordination_type sf)
		: initial(o.source_operator, std::move(sf))
	{
	}

	template<class Subscriber>
	void on_subscribe(Subscriber scbr) const {
		static_assert(is_subscriber<Subscriber>::value, "subscribe must be passed a subscriber");

		typedef Subscriber output_type;

		struct amb_state_type
			: public std::enable_shared_from_this<amb_state_type>
			, public values
		{
			amb_state_type(values i, coordinator_type coor, output_type oarg)
			: values(i)
			, source(i.source_operator)
			, coordinator(std::move(coor))
			, out(std::move(oarg))
			, pendingObservables(0)
			, firstEmitted(false)
			{
			}
			observable<source_value_type, source_operator_type> source;
			coordinator_type coordinator;
			output_type out;
			int pendingObservables;
			bool firstEmitted;
			std::vector<composite_subscription> innerSubscriptions;
		};

		auto coordinator = initial.coordination.create_coordinator(scbr.get_subscription());

		// take a copy of the values for each subscription
		auto state = std::make_shared<amb_state_type>(initial, std::move(coordinator),
std::move(scbr));

		composite_subscription outercs;

		// when the out observer is unsubscribed all the
		// inner subscriptions are unsubscribed as well
		state->out.add(outercs);

		auto source = on_exception(
			[&](){return state->coordinator.in(state->source); },
			state->out);
		if (source.empty()) {
			return;
		}
```

```
                                        // this subscribe does not share the observer subscription
                                        // so that when it is unsubscribed the observer can be called
                                        // until the inner subscriptions have finished
                                        auto sink = make_subscriber<source_value_type>(
                                                state->out,
                                                outercs,
                                                // on_next
                                                [state](source_value_type st) {

                                                if (state->firstEmitted)
                                                        return;

                                                composite_subscription innercs;

                                                state->innerSubscriptions.push_back(innercs);

                                                // when the out observer is unsubscribed all the
                                                // inner subscriptions are unsubscribed as well
                                                auto innercstoken = state->out.add(innercs);

                                                innercs.add(make_subscription([state, innercstoken](){
                                                        state->out.remove(innercstoken);
                                                }));

                                                auto selectedSource = state->coordinator.in(st);

                                                auto current_id = state->pendingObservables++;

                                                // this subscribe does not share the source subscription
                                                // so that when it is unsubscribed the source will continue
                                                auto sinkInner = make_subscriber<value_type>(
                                                        state->out,
                                                        innercs,
                                                        // on_next
                                                        [state, st, current_id](value_type ct) {
                                                        state->out.on_next(std::move(ct));
                                                        if (!state->firstEmitted) {
                                                                state->firstEmitted = true;
                                                                auto do_unsubscribe = [](composite_subscription cs) {
                                                                        cs.unsubscribe();
                                                                };
                                                                std::for_each(state->innerSubscriptions.begin(), state-
>innerSubscriptions.begin() + current_id, do_unsubscribe);
                                                                std::for_each(state->innerSubscriptions.begin() +
current_id + 1, state->innerSubscriptions.end(), do_unsubscribe);
                                                        }
                                                },
                                                        // on_error
                                                        [state](std::exception_ptr e) {
                                                        state->out.on_error(e);
                                                },
                                                        //on_completed
                                                        [state](){
                                                        state->out.on_completed();
                                                }
                                                );

                                                auto selectedSinkInner = state->coordinator.out(sinkInner);
                                                selectedSource.subscribe(std::move(selectedSinkInner));
                                        },
                                                // on_error
                                                [state](std::exception_ptr e) {
                                                state->out.on_error(e);
                                        },
                                                // on_completed
                                                [state]() {
                                                if (state->pendingObservables == 0) {
                                                        state->out.on_completed();
                                                }
                                        }
                                        );
                                        auto selectedSink = on_exception(
                                                [&](){return state->coordinator.out(sink); },
                                                state->out);
                                        if (selectedSink.empty()) {
                                                return;
                                        }
                                        source->subscribe(std::move(selectedSink.get()));
                                }
                        };

                        template<class Coordination>
```

```
                                class amb_factory
                                {
                                        typedef rxu::decay_t<Coordination> coordination_type;

                                        coordination_type coordination;
                                public:
                                        amb_factory(coordination_type sf)
                                                : coordination(std::move(sf))
                                        {
                                        }

                                        template<class Observable>
                                        auto operator()(Observable source)
                                                ->      observable<rxu::value_type_t<amb<rxu::value_type_t<Observable>, Observable,
Coordination>>, amb<rxu::value_type_t<Observable>, Observable, Coordination>> {
                                                return  observable<rxu::value_type_t<amb<rxu::value_type_t<Observable>, Observable,
Coordination>>, amb<rxu::value_type_t<Observable>, Observable, Coordination>>(
                                                                amb<rxu::value_type_t<Observable>, Observable,
Coordination>(std::move(source), coordination));
                                        }
                                };

                        }

                        template<class Coordination>
                        auto amb(Coordination&& sf)
                                ->      detail::amb_factory<Coordination> {
                                return  detail::amb_factory<Coordination>(std::forward<Coordination>(sf));
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_BUFFER_COUNT_HPP)
#define RXCPP_OPERATORS_RX_BUFFER_COUNT_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {


                        template<class T>
                        struct buffer_count
                        {
                                typedef rxu::decay_t<T> source_value_type;
                                struct buffer_count_values
                                {
                                        buffer_count_values(int c, int s)
                                        : count(c)
                                        , skip(s)
                                        {
                                        }
                                        int count;
                                        int skip;
                                };

                                buffer_count_values initial;

                                buffer_count(int count, int skip)
                                        : initial(count, skip)
                                {
                                }

                                template<class Subscriber>
                                struct buffer_count_observer : public buffer_count_values
                                {
                                        typedef buffer_count_observer<Subscriber> this_type;
                                        typedef std::vector<T> value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<value_type, this_type> observer_type;
                                        dest_type dest;
                                        mutable int cursor;
                                        mutable std::deque<value_type> chunks;
```

```
                                                buffer_count_observer(dest_type d, buffer_count_values v)
                                                        : buffer_count_values(v)
                                                        , dest(std::move(d))
                                                        , cursor(0)
                                                {
                                                }
                                                void on_next(T v) const {
                                                        if (cursor++ % this->skip == 0) {
                                                                chunks.emplace_back();
                                                        }
                                                        for (auto& chunk : chunks) {
                                                                chunk.push_back(v);
                                                        }
                                                        while (!chunks.empty() && int(chunks.front().size()) == this->count) {
                                                                dest.on_next(std::move(chunks.front()));
                                                                chunks.pop_front();
                                                        }
                                                }
                                                void on_error(std::exception_ptr e) const {
                                                        dest.on_error(e);
                                                }
                                                void on_completed() const {
                                                        auto done = on_exception(
                                                                [&](){
                                                                while (!chunks.empty()) {
                                                                        dest.on_next(std::move(chunks.front()));
                                                                        chunks.pop_front();
                                                                }
                                                                return true;
                                                        },
                                                                dest);
                                                        if (done.empty()) {
                                                                return;
                                                        }
                                                        dest.on_completed();
                                                }

                                                static subscriber<T, observer<T, this_type>> make(dest_type d, buffer_count_values v) {
                                                        auto cs = d.get_subscription();
                                                        return make_subscriber<T>(std::move(cs), this_type(std::move(d),
std::move(v)));
                                                }
                                        };

                                        template<class Subscriber>
                                        auto operator()(Subscriber dest) const
                                                -> decltype(buffer_count_observer<Subscriber>::make(std::move(dest), initial)) {
                                                return     buffer_count_observer<Subscriber>::make(std::move(dest), initial);
                                        }
                                };

                                class buffer_count_factory
                                {
                                        int count;
                                        int skip;
                                public:
                                        buffer_count_factory(int c, int s) : count(c), skip(s) {}
                                        template<class Observable>
                                        auto operator()(Observable&& source)
                                                -> decltype(source.template
lift<std::vector<rxu::value_type_t<rxu::decay_t<Observable>>>>(buffer_count<rxu::value_type_t<rxu::decay_t<Observable>>>(count, skip))) {
                                                return     source.template
lift<std::vector<rxu::value_type_t<rxu::decay_t<Observable>>>>(buffer_count<rxu::value_type_t<rxu::decay_t<Observable>>>(count, skip));
                                        }
                                };

                        }

                        inline auto buffer(int count)
                                ->     detail::buffer_count_factory {
                                return  detail::buffer_count_factory(count, count);
                        }
                        inline auto buffer(int count, int skip)
                                ->     detail::buffer_count_factory {
                                return  detail::buffer_count_factory(count, skip);
                        }

                }

}

#endif
```

```
#if !defined(RXCPP_OPERATORS_RX_BUFFER_WITH_TIME_HPP)
#define RXCPP_OPERATORS_RX_BUFFER_WITH_TIME_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace operators {

		namespace detail {

			template<class T, class Duration, class Coordination>
			struct buffer_with_time
			{
				static_assert(std::is_convertible<Duration, rxsc::scheduler::clock_type::duration>::value, "Duration parameter must convert to rxsc::scheduler::clock_type::duration");
				static_assert(is_coordination<Coordination>::value, "Coordination parameter must satisfy the requirements for a Coordination");

				typedef rxu::decay_t<T> source_value_type;
				typedef rxu::decay_t<Coordination> coordination_type;
				typedef typename coordination_type::coordinator_type coordinator_type;
				typedef rxu::decay_t<Duration> duration_type;

				struct buffer_with_time_values
				{
					buffer_with_time_values(duration_type p, duration_type s, coordination_type c)
					: period(p)
					, skip(s)
					, coordination(c)
					{
					}
					duration_type period;
					duration_type skip;
					coordination_type coordination;
				};
				buffer_with_time_values initial;

				buffer_with_time(duration_type period, duration_type skip, coordination_type coordination)
					: initial(period, skip, coordination)
				{
				}

				template<class Subscriber>
				struct buffer_with_time_observer
				{
					typedef buffer_with_time_observer<Subscriber> this_type;
					typedef std::vector<T> value_type;
					typedef rxu::decay_t<Subscriber> dest_type;
					typedef observer<value_type, this_type> observer_type;

					struct buffer_with_time_subscriber_values : public buffer_with_time_values
					{
						buffer_with_time_subscriber_values(composite_subscription cs, dest_type d, buffer_with_time_values v, coordinator_type c)
						: buffer_with_time_values(v)
						, cs(std::move(cs))
						, dest(std::move(d))
						, coordinator(std::move(c))
						, worker(coordinator.get_worker())
						, expected(worker.now())
						{
						}
						composite_subscription cs;
						dest_type dest;
						coordinator_type coordinator;
						rxsc::worker worker;
						mutable std::deque<value_type> chunks;
						rxsc::scheduler::clock_type::time_point expected;
					};
					std::shared_ptr<buffer_with_time_subscriber_values> state;

					buffer_with_time_observer(composite_subscription cs, dest_type d, buffer_with_time_values v, coordinator_type c)
						: state(std::make_shared<buffer_with_time_subscriber_values>(buffer_with_time_subscriber_values(std::move(cs), std::move(d), v, std::move(c))))
					{
						auto localState = state;
```

```
                auto disposer = [=](const rxsc::schedulable&){
                        localState->cs.unsubscribe();
                        localState->dest.unsubscribe();
                        localState->worker.unsubscribe();
                };
                auto selectedDisposer = on_exception(
                        [&](){return localState->coordinator.act(disposer); },
                        localState->dest);
                if (selectedDisposer.empty()) {
                        return;
                }

                localState->dest.add([=](){
                        localState->worker.schedule(selectedDisposer.get());
                });
                localState->cs.add([=](){
                        localState->worker.schedule(selectedDisposer.get());
                });

                //
                // The scheduler is FIFO for any time T. Since the observer is scheduling
                // on_next/on_error/oncompleted the timed schedule calls must be resheduled
                // when they occur to ensure that production happens after
on_next/on_error/oncompleted
                //

                auto produce_buffer = [localState](const rxsc::schedulable&) {
                        localState->dest.on_next(std::move(localState->chunks.front()));
                        localState->chunks.pop_front();
                };
                auto selectedProduce = on_exception(
                        [&](){return localState->coordinator.act(produce_buffer); },
                        localState->dest);
                if (selectedProduce.empty()) {
                        return;
                }

                auto create_buffer = [localState, selectedProduce](const rxsc::schedulable&) {
                        localState->chunks.emplace_back();
                        auto produce_at = localState->expected + localState->period;
                        localState->expected += localState->skip;
                        localState->worker.schedule(produce_at, [localState,
selectedProduce](const rxsc::schedulable&) {
                                localState->worker.schedule(selectedProduce.get());
                        });
                };
                auto selectedCreate = on_exception(
                        [&](){return localState->coordinator.act(create_buffer); },
                        localState->dest);
                if (selectedCreate.empty()) {
                        return;
                }

                state->worker.schedule_periodically(
                        state->expected,
                        state->skip,
                        [localState, selectedCreate](const rxsc::schedulable&) {
                        localState->worker.schedule(selectedCreate.get());
                });
        }
        void on_next(T v) const {
                auto localState = state;
                auto work = [v, localState](const rxsc::schedulable&){
                        for (auto& chunk : localState->chunks) {
                                chunk.push_back(v);
                        }
                };
                auto selectedWork = on_exception(
                        [&](){return localState->coordinator.act(work); },
                        localState->dest);
                if (selectedWork.empty()) {
                        return;
                }
                localState->worker.schedule(selectedWork.get());
        }
        void on_error(std::exception_ptr e) const {
                auto localState = state;
                auto work = [e, localState](const rxsc::schedulable&){
                        localState->dest.on_error(e);
                };
                auto selectedWork = on_exception(
                        [&](){return localState->coordinator.act(work); },
```

```
                                                                localState->dest);
                                                        if (selectedWork.empty()) {
                                                                return;
                                                        }
                                                        localState->worker.schedule(selectedWork.get());
                                                }
                                                void on_completed() const {
                                                        auto localState = state;
                                                        auto work = [localState](const rxsc::schedulable&){
                                                                on_exception(
                                                                        [&](){
                                                                        while (!localState->chunks.empty()) {
                                                                                localState->dest.on_next(std::move(localState->chunks.front()));
                                                                                localState->chunks.pop_front();
                                                                        }
                                                                        return true;
                                                                },
                                                                        localState->dest);
                                                                localState->dest.on_completed();
                                                        };
                                                        auto selectedWork = on_exception(
                                                                [&](){return localState->coordinator.act(work); },
                                                                localState->dest);
                                                        if (selectedWork.empty()) {
                                                                return;
                                                        }
                                                        localState->worker.schedule(selectedWork.get());
                                                }

                                                static subscriber<T, observer<T, this_type>> make(dest_type d, buffer_with_time_values v) {
                                                        auto cs = composite_subscription();
                                                        auto coordinator = v.coordination.create_coordinator();

                                                        return make_subscriber<T>(cs, this_type(cs, std::move(d), std::move(v), std::move(coordinator)));
                                                }
                                        };

                                        template<class Subscriber>
                                        auto operator()(Subscriber dest) const
                                                -> decltype(buffer_with_time_observer<Subscriber>::make(std::move(dest), initial)) {
                                                return     buffer_with_time_observer<Subscriber>::make(std::move(dest), initial);
                                        }
                                };

                                template<class Duration, class Coordination>
                                class buffer_with_time_factory
                                {
                                        typedef rxu::decay_t<Duration> duration_type;
                                        typedef rxu::decay_t<Coordination> coordination_type;

                                        duration_type period;
                                        duration_type skip;
                                        coordination_type coordination;
                                public:
                                        buffer_with_time_factory(duration_type p, duration_type s, coordination_type c) : period(p), skip(s), coordination(c) {}
                                        template<class Observable>
                                        auto operator()(Observable&& source)
                                                -> decltype(source.template lift<std::vector<rxu::value_type_t<rxu::decay_t<Observable>>>>(buffer_with_time<rxu::value_type_t<rxu::decay_t<Observable>>, Duration, Coordination>(period, skip, coordination))) {
                                                return     source.template lift<std::vector<rxu::value_type_t<rxu::decay_t<Observable>>>>(buffer_with_time<rxu::value_type_t<rxu::decay_t<Observable>>, Duration, Coordination>(period, skip, coordination));
                                        }
                                };

                        }

                        template<class Duration, class Coordination>
                        inline auto buffer_with_time(Duration period, Coordination coordination)
                                ->      detail::buffer_with_time_factory<Duration, Coordination> {
                                return  detail::buffer_with_time_factory<Duration, Coordination>(period, period, coordination);
                        }

                        template<class Duration, class Coordination>
                        inline auto buffer_with_time(Duration period, Duration skip, Coordination coordination)
                                ->      detail::buffer_with_time_factory<Duration, Coordination> {
                                return  detail::buffer_with_time_factory<Duration, Coordination>(period, skip, coordination);
```

```
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_BUFFER_WITH_TIME_OR_COUNT_HPP)
#define RXCPP_OPERATORS_RX_BUFFER_WITH_TIME_OR_COUNT_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Duration, class Coordination>
                        struct buffer_with_time_or_count
                        {
                                static_assert(std::is_convertible<Duration, rxsc::scheduler::clock_type::duration>::value, "Duration
parameter must convert to rxsc::scheduler::clock_type::duration");
                                static_assert(is_coordination<Coordination>::value, "Coordination parameter must satisfy the
requirements for a Coordination");

                                typedef rxu::decay_t<T> source_value_type;
                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef typename coordination_type::coordinator_type coordinator_type;
                                typedef rxu::decay_t<Duration> duration_type;

                                struct buffer_with_time_or_count_values
                                {
                                        buffer_with_time_or_count_values(duration_type p, int n, coordination_type c)
                                        : period(p)
                                        , count(n)
                                        , coordination(c)
                                        {
                                        }
                                        duration_type period;
                                        int count;
                                        coordination_type coordination;
                                };
                                buffer_with_time_or_count_values initial;

                                buffer_with_time_or_count(duration_type period, int count, coordination_type coordination)
                                        : initial(period, count, coordination)
                                {
                                }

                                template<class Subscriber>
                                struct buffer_with_time_or_count_observer
                                {
                                        typedef buffer_with_time_or_count_observer<Subscriber> this_type;
                                        typedef std::vector<T> value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<value_type, this_type> observer_type;

                                        struct buffer_with_time_or_count_subscriber_values : public
buffer_with_time_or_count_values
                                        {
                                                buffer_with_time_or_count_subscriber_values(composite_subscription cs,
dest_type d, buffer_with_time_or_count_values v, coordinator_type c)
                                                : buffer_with_time_or_count_values(std::move(v))
                                                , cs(std::move(cs))
                                                , dest(std::move(d))
                                                , coordinator(std::move(c))
                                                , worker(coordinator.get_worker())
                                                , chunk_id(0)
                                                {
                                                }
                                                composite_subscription cs;
                                                dest_type dest;
                                                coordinator_type coordinator;
                                                rxsc::worker worker;
                                                mutable int chunk_id;
                                                mutable value_type chunk;
                                        };
                                        typedef std::shared_ptr<buffer_with_time_or_count_subscriber_values> state_type;
                                        state_type state;
```

```
                buffer_with_time_or_count_observer(composite_subscription cs, dest_type d,
buffer_with_time_or_count_values v, coordinator_type c)
                    :
state(std::make_shared<buffer_with_time_or_count_subscriber_values>(buffer_with_time_or_count_subscriber_values(std::move(cs),
std::move(d), std::move(v), std::move(c))))
                {
                    auto new_id = state->chunk_id;
                    auto produce_time = state->worker.now() + state->period;
                    auto localState = state;

                    auto disposer = [=](const rxsc::schedulable&){
                        localState->cs.unsubscribe();
                        localState->dest.unsubscribe();
                        localState->worker.unsubscribe();
                    };
                    auto selectedDisposer = on_exception(
                        [&](){return localState->coordinator.act(disposer); },
                        localState->dest);
                    if (selectedDisposer.empty()) {
                        return;
                    }

                    localState->dest.add([=](){
                        localState->worker.schedule(selectedDisposer.get());
                    });
                    localState->cs.add([=](){
                        localState->worker.schedule(selectedDisposer.get());
                    });

                    //
                    // The scheduler is FIFO for any time T. Since the observer is scheduling
                    // on_next/on_error/oncompleted the timed schedule calls must be resheduled
                    // when they occur to ensure that production happens after
on_next/on_error/oncompleted
                    //

                    localState->worker.schedule(produce_time, [new_id, produce_time,
localState](const rxsc::schedulable&){
                        localState->worker.schedule(produce_buffer(new_id, produce_time,
localState));
                    });
                }

                static std::function<void(const rxsc::schedulable&)> produce_buffer(int id,
rxsc::scheduler::clock_type::time_point expected, state_type state) {
                    auto produce = [id, expected, state](const rxsc::schedulable&) {
                        if (id != state->chunk_id)
                            return;

                        state->dest.on_next(state->chunk);
                        state->chunk.resize(0);
                        auto new_id = ++state->chunk_id;
                        auto produce_time = expected + state->period;
                        state->worker.schedule(produce_time, [new_id, produce_time,
state](const rxsc::schedulable&){
                            state->worker.schedule(produce_buffer(new_id,
produce_time, state));
                        });
                    };

                    auto selectedProduce = on_exception(
                        [&](){return state->coordinator.act(produce); },
                        state->dest);
                    if (selectedProduce.empty()) {
                        return std::function<void(const rxsc::schedulable&)>();
                    }

                    return std::function<void(const rxsc::schedulable&)>(selectedProduce.get());
                }

                void on_next(T v) const {
                    auto localState = state;
                    auto work = [v, localState](const rxsc::schedulable& self){
                        localState->chunk.push_back(v);
                        if (int(localState->chunk.size()) == localState->count) {
                            produce_buffer(localState->chunk_id, localState-
>worker.now(), localState)(self);
                        }
                    };
                    auto selectedWork = on_exception(
                        [&](){return localState->coordinator.act(work); },
```

```
                                                                localState->dest);
                                                        if (selectedWork.empty()) {
                                                                return;
                                                        }
                                                        localState->worker.schedule(selectedWork.get());
                                                }
                                                void on_error(std::exception_ptr e) const {
                                                        auto localState = state;
                                                        auto work = [e, localState](const rxsc::schedulable&){
                                                                localState->dest.on_error(e);
                                                        };
                                                        auto selectedWork = on_exception(
                                                                [&](){return localState->coordinator.act(work); },
                                                                localState->dest);
                                                        if (selectedWork.empty()) {
                                                                return;
                                                        }
                                                        localState->worker.schedule(selectedWork.get());
                                                }
                                                void on_completed() const {
                                                        auto localState = state;
                                                        auto work = [localState](const rxsc::schedulable&){
                                                                localState->dest.on_next(localState->chunk);
                                                                localState->dest.on_completed();
                                                        };
                                                        auto selectedWork = on_exception(
                                                                [&](){return localState->coordinator.act(work); },
                                                                localState->dest);
                                                        if (selectedWork.empty()) {
                                                                return;
                                                        }
                                                        localState->worker.schedule(selectedWork.get());
                                                }

                                                static subscriber<T, observer<T, this_type>> make(dest_type d,
buffer_with_time_or_count_values v) {
                                                        auto cs = composite_subscription();
                                                        auto coordinator = v.coordination.create_coordinator();

                                                        return make_subscriber<T>(cs, this_type(cs, std::move(d), std::move(v),
std::move(coordinator)));
                                                }
                                        };

                                        template<class Subscriber>
                                        auto operator()(Subscriber dest) const
                                                -> decltype(buffer_with_time_or_count_observer<Subscriber>::make(std::move(dest),
initial)) {
                                                return     buffer_with_time_or_count_observer<Subscriber>::make(std::move(dest),
initial);
                                        }
                                };

                                template<class Duration, class Coordination>
                                class buffer_with_time_or_count_factory
                                {
                                        typedef rxu::decay_t<Duration> duration_type;
                                        typedef rxu::decay_t<Coordination> coordination_type;

                                        duration_type period;
                                        duration_type skip;
                                        coordination_type coordination;
                                public:
                                        buffer_with_time_or_count_factory(duration_type p, duration_type s, coordination_type c) :
period(p), skip(s), coordination(c) {}
                                        template<class Observable>
                                        auto operator()(Observable&& source)
                                                -> decltype(source.template
lift<std::vector<rxu::value_type_t<rxu::decay_t<Observable>>>>(buffer_with_time_or_count<rxu::value_type_t<rxu::decay_t<Observable>>,
Duration, Coordination>(period, skip, coordination))) {
                                                return     source.template
lift<std::vector<rxu::value_type_t<rxu::decay_t<Observable>>>>(buffer_with_time_or_count<rxu::value_type_t<rxu::decay_t<Observable>>,
Duration, Coordination>(period, skip, coordination));
                                        }
                                };

                        }

                        template<class Duration, class Coordination>
                        inline auto buffer_with_time_or_count(Duration period, int count, Coordination coordination)
                                ->      detail::buffer_with_time_or_count_factory<Duration, Coordination> {
                                return  detail::buffer_with_time_or_count_factory<Duration, Coordination>(period, count, coordination);
```

```
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_CONCAT_HPP)
#define RXCPP_OPERATORS_RX_CONCAT_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Observable, class Coordination>
                        struct concat
                                : public operator_base<rxu::value_type_t<rxu::decay_t<T>>>
                        {
                                typedef concat<T, Observable, Coordination> this_type;

                                typedef rxu::decay_t<T> source_value_type;
                                typedef rxu::decay_t<Observable> source_type;
                                typedef rxu::decay_t<Coordination> coordination_type;

                                typedef typename coordination_type::coordinator_type coordinator_type;

                                typedef typename source_type::source_operator_type source_operator_type;
                                typedef source_value_type collection_type;
                                typedef typename collection_type::value_type value_type;

                                struct values
                                {
                                        values(source_operator_type o, coordination_type sf)
                                        : source_operator(std::move(o))
                                        , coordination(std::move(sf))
                                        {
                                        }
                                        source_operator_type source_operator;
                                        coordination_type coordination;
                                };
                                values initial;

                                concat(const source_type& o, coordination_type sf)
                                        : initial(o.source_operator, std::move(sf))
                                {
                                }

                                template<class Subscriber>
                                void on_subscribe(Subscriber scbr) const {
                                        static_assert(is_subscriber<Subscriber>::value, "subscribe must be passed a subscriber");

                                        typedef Subscriber output_type;

                                        struct concat_state_type
                                                : public std::enable_shared_from_this<concat_state_type>
                                                , public values
                                        {
                                                concat_state_type(values i, coordinator_type coor, output_type oarg)
                                                : values(i)
                                                , source(i.source_operator)
                                                , sourceLifetime(composite_subscription::empty())
                                                , collectionLifetime(composite_subscription::empty())
                                                , coordinator(std::move(coor))
                                                , out(std::move(oarg))
                                                {
                                                }

                                                void subscribe_to(collection_type st)
                                                {
                                                        auto state = this->shared_from_this();

                                                        collectionLifetime = composite_subscription();

                                                        // when the out observer is unsubscribed all the
                                                        // inner subscriptions are unsubscribed as well
                                                        auto innercstoken = state->out.add(collectionLifetime);
```

```
                                                collectionLifetime.add(make_subscription([state, innercstoken](){
                                                        state->out.remove(innercstoken);
                                                }));

                                                auto selectedSource = on_exception(
                                                        [&](){return state->coordinator.in(std::move(st)); },
                                                        state->out);
                                                if (selectedSource.empty()) {
                                                        return;
                                                }

                                                // this subscribe does not share the out subscription
                                                // so that when it is unsubscribed the out will continue
                                                auto sinkInner = make_subscriber<value_type>(
                                                        state->out,
                                                        collectionLifetime,
                                                        // on_next
                                                        [state, st](value_type ct) {
                                                        state->out.on_next(ct);
                                                },
                                                        // on_error
                                                        [state](std::exception_ptr e) {
                                                        state->out.on_error(e);
                                                },
                                                        //on_completed
                                                        [state](){
                                                        if (!state->selectedCollections.empty()) {
                                                                auto value = state->selectedCollections.front();
                                                                state->selectedCollections.pop_front();
                                                                state->collectionLifetime.unsubscribe();
                                                                state->subscribe_to(value);
                                                        }
                                                        else if (!state->sourceLifetime.is_subscribed()) {
                                                                state->out.on_completed();
                                                        }
                                                }
                                                );
                                                auto selectedSinkInner = on_exception(
                                                        [&](){return state->coordinator.out(sinkInner); },
                                                        state->out);
                                                if (selectedSinkInner.empty()) {
                                                        return;
                                                }
                                                selectedSource->subscribe(std::move(selectedSinkInner.get()));
                                        }
                                        observable<source_value_type, source_operator_type> source;
                                        composite_subscription sourceLifetime;
                                        composite_subscription collectionLifetime;
                                        std::deque<collection_type> selectedCollections;
                                        coordinator_type coordinator;
                                        output_type out;
                };

                auto coordinator = initial.coordination.create_coordinator(scbr.get_subscription());

                // take a copy of the values for each subscription
                auto state = std::make_shared<concat_state_type>(initial, std::move(coordinator),
std::move(scbr));


                state->sourceLifetime = composite_subscription();

                // when the out observer is unsubscribed all the
                // inner subscriptions are unsubscribed as well
                state->out.add(state->sourceLifetime);

                auto source = on_exception(
                        [&](){return state->coordinator.in(state->source); },
                        state->out);
                if (source.empty()) {
                        return;
                }

                // this subscribe does not share the observer subscription
                // so that when it is unsubscribed the observer can be called
                // until the inner subscriptions have finished
                auto sink = make_subscriber<collection_type>(
                        state->out,
                        state->sourceLifetime,
                        // on_next
                        [state](collection_type st) {
                        if (state->collectionLifetime.is_subscribed()) {
```

```
                                                                        state->selectedCollections.push_back(st);
                                                                }
                                                                else if (state->selectedCollections.empty()) {
                                                                        state->subscribe_to(st);
                                                                }
                                                        },
                                                                // on_error
                                                                [state](std::exception_ptr e) {
                                                                state->out.on_error(e);
                                                        },
                                                                // on_completed
                                                                [state]() {
                                                                if (!state->collectionLifetime.is_subscribed() && state->selectedCollections.empty()) {
                                                                        state->out.on_completed();
                                                                }
                                                        }
                                                        );
                                                        auto selectedSink = on_exception(
                                                                [&](){return state->coordinator.out(sink); },
                                                                state->out);
                                                        if (selectedSink.empty()) {
                                                                return;
                                                        }
                                                        source->subscribe(std::move(selectedSink.get()));
                                                }
                                        };

                                        template<class Coordination>
                                        class concat_factory
                                        {
                                                typedef rxu::decay_t<Coordination> coordination_type;

                                                coordination_type coordination;
                                        public:
                                                concat_factory(coordination_type sf)
                                                        : coordination(std::move(sf))
                                                {
                                                }

                                                template<class Observable>
                                                auto operator()(Observable source)
                                                        -> observable<rxu::value_type_t<concat<rxu::value_type_t<Observable>, Observable, Coordination>>, concat<rxu::value_type_t<Observable>, Observable, Coordination>> {
                                                        return observable<rxu::value_type_t<concat<rxu::value_type_t<Observable>, Observable, Coordination>>, concat<rxu::value_type_t<Observable>, Observable, Coordination>>(
                                                                concat<rxu::value_type_t<Observable>, Observable, Coordination>(std::move(source), coordination));
                                                }
                                        };

                        }

                        inline auto concat()
                                -> detail::concat_factory<identity_one_worker> {
                                return detail::concat_factory<identity_one_worker>(identity_current_thread());
                        }

                        template<class Coordination, class Check = typename std::enable_if<is_coordination<Coordination>::value>::type>
                        auto concat(Coordination&& sf)
                                -> detail::concat_factory<Coordination> {
                                return detail::concat_factory<Coordination>(std::forward<Coordination>(sf));
                        }

                        template<class O0, class... ON, class Check = typename std::enable_if<is_observable<O0>::value>::type>
                        auto concat(O0&& o0, ON&&... on)
                                -> detail::concat_factory<identity_one_worker> {
                                return detail::concat_factory<identity_one_worker>(identity_current_thread())(from(std::forward<O0>(o0), std::forward<ON>(on)...));
                        }

                        template<class Coordination, class O0, class... ON,
                        class CheckC = typename std::enable_if<is_coordination<Coordination>::value>::type,
                        class CheckO = typename std::enable_if<is_observable<O0>::value>::type>
                                auto concat(Coordination&& sf, O0&& o0, ON&&... on)
                                -> detail::concat_factory<Coordination> {
                                        return detail::concat_factory<Coordination>(std::forward<Coordination>(sf))(from(std::forward<O0>(o0), std::forward<ON>(on)...));
                                }

        }
```

```
}

#endif


#if !defined(RXCPP_OPERATORS_RX_CONCATMAP_HPP)
#define RXCPP_OPERATORS_RX_CONCATMAP_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace operators {

		namespace detail {

			template<class Observable, class CollectionSelector, class ResultSelector, class Coordination>
			struct concat_traits {
				typedef rxu::decay_t<Observable> source_type;
				typedef rxu::decay_t<CollectionSelector> collection_selector_type;
				typedef rxu::decay_t<ResultSelector> result_selector_type;
				typedef rxu::decay_t<Coordination> coordination_type;

				typedef typename source_type::value_type source_value_type;

				struct tag_not_valid {};
				template<class CV, class CCS>
				static auto collection_check(int) -> decltype((*(CCS*)nullptr)(*(CV*)nullptr));
				template<class CV, class CCS>
				static tag_not_valid collection_check(...);

				static_assert(!std::is_same<decltype(collection_check<source_value_type,
collection_selector_type>(0)), tag_not_valid>::value, "concat_map CollectionSelector must be a function with the signature
observable(concat_map::source_value_type)");

				typedef decltype((*(collection_selector_type*)nullptr)((*(source_value_type*)nullptr)))
collection_type;

				//#if _MSC_VER >= 1900
				static_assert(is_observable<collection_type>::value, "concat_map CollectionSelector must return an
observable");
				//#endif

				typedef typename collection_type::value_type collection_value_type;

				template<class CV, class CCV, class CRS>
				static auto result_check(int) -> decltype((*(CRS*)nullptr)(*(CV*)nullptr, *(CCV*)nullptr));
				template<class CV, class CCV, class CRS>
				static tag_not_valid result_check(...);

				static_assert(!std::is_same<decltype(result_check<source_value_type, collection_value_type,
result_selector_type>(0)), tag_not_valid>::value, "concat_map ResultSelector must be a function with the signature
concat_map::value_type(concat_map::source_value_type, concat_map::collection_value_type)");

				typedef rxu::decay_t<decltype((*(result_selector_type*)nullptr)(*(source_value_type*)nullptr,
*(collection_value_type*)nullptr))> value_type;
			};

			template<class Observable, class CollectionSelector, class ResultSelector, class Coordination>
			struct concat_map
				: public operator_base<rxu::value_type_t<concat_traits<Observable, CollectionSelector,
ResultSelector, Coordination>>>
			{
				typedef concat_map<Observable, CollectionSelector, ResultSelector, Coordination> this_type;
				typedef concat_traits<Observable, CollectionSelector, ResultSelector, Coordination> traits;

				typedef typename traits::source_type source_type;
				typedef typename traits::collection_selector_type collection_selector_type;
				typedef typename traits::result_selector_type result_selector_type;

				typedef typename traits::source_value_type source_value_type;
				typedef typename traits::collection_type collection_type;
				typedef typename traits::collection_value_type collection_value_type;

				typedef typename traits::coordination_type coordination_type;
				typedef typename coordination_type::coordinator_type coordinator_type;

				struct values
				{
					values(source_type o, collection_selector_type s, result_selector_type rs, coordination_type
sf)
					: source(std::move(o))
```

```
                , selectCollection(std::move(s))
                , selectResult(std::move(rs))
                , coordination(std::move(sf))
                {
                }
                source_type source;
                collection_selector_type selectCollection;
                result_selector_type selectResult;
                coordination_type coordination;
        private:
                values& operator=(const values&)RXCPP_DELETE;
        };
        values initial;

        concat_map(source_type o, collection_selector_type s, result_selector_type rs, coordination_type sf)
                : initial(std::move(o), std::move(s), std::move(rs), std::move(sf))
        {
        }

        template<class Subscriber>
        void on_subscribe(Subscriber scbr) const {
                static_assert(is_subscriber<Subscriber>::value, "subscribe must be passed a subscriber");

                typedef Subscriber output_type;

                struct concat_map_state_type
                        : public std::enable_shared_from_this<concat_map_state_type>
                        , public values
                {
                        concat_map_state_type(values i, coordinator_type coor, output_type oarg)
                        : values(std::move(i))
                        , sourceLifetime(composite_subscription::empty())
                        , collectionLifetime(composite_subscription::empty())
                        , coordinator(std::move(coor))
                        , out(std::move(oarg))
                        {
                        }

                        void subscribe_to(source_value_type st)
                        {
                                auto state = this->shared_from_this();

                                auto selectedCollection = on_exception(
                                        [&](){return state->selectCollection(st); },
                                        state->out);
                                if (selectedCollection.empty()) {
                                        return;
                                }

                                collectionLifetime = composite_subscription();

                                // when the out observer is unsubscribed all the
                                // inner subscriptions are unsubscribed as well
                                auto innercstoken = state->out.add(collectionLifetime);

                                collectionLifetime.add(make_subscription([state, innercstoken](){
                                        state->out.remove(innercstoken);
                                }));

                                auto selectedSource = on_exception(
                                        [&](){return state-
>coordinator.in(selectedCollection.get()); },
                                        state->out);
                                if (selectedSource.empty()) {
                                        return;
                                }

                                // this subscribe does not share the source subscription
                                // so that when it is unsubscribed the source will continue
                                auto sinkInner = make_subscriber<collection_value_type>(
                                        state->out,
                                        collectionLifetime,
                                        // on_next
                                        [state, st](collection_value_type ct) {
                                        auto selectedResult = state->selectResult(st,
std::move(ct));
                                        state->out.on_next(std::move(selectedResult));
                                },
                                        // on_error
                                        [state](std::exception_ptr e) {
                                        state->out.on_error(e);
                                },
```

```
                                                                        //on_completed
                                                                        [state](){
                                                                        if (!state->selectedCollections.empty()) {
                                                                                auto value = state->selectedCollections.front();
                                                                                state->selectedCollections.pop_front();
                                                                                state->collectionLifetime.unsubscribe();
                                                                                state->subscribe_to(value);
                                                                        }
                                                                        else if (!state->sourceLifetime.is_subscribed()) {
                                                                                state->out.on_completed();
                                                                        }
                                                                }
                                                                );
                                                                auto selectedSinkInner = on_exception(
                                                                        [&](){return state->coordinator.out(sinkInner); },
                                                                        state->out);
                                                                if (selectedSinkInner.empty()) {
                                                                        return;
                                                                }
                                                                selectedSource->subscribe(std::move(selectedSinkInner.get()));
                                                        }
                                                        composite_subscription sourceLifetime;
                                                        composite_subscription collectionLifetime;
                                                        std::deque<source_value_type> selectedCollections;
                                                        coordinator_type coordinator;
                                                        output_type out;
                                                };

                                                auto coordinator = initial.coordination.create_coordinator(scbr.get_subscription());

                                                // take a copy of the values for each subscription
                                                auto state = std::make_shared<concat_map_state_type>(initial, std::move(coordinator),
std::move(scbr));


                                                state->sourceLifetime = composite_subscription();

                                                // when the out observer is unsubscribed all the
                                                // inner subscriptions are unsubscribed as well
                                                state->out.add(state->sourceLifetime);

                                                auto source = on_exception(
                                                        [&](){return state->coordinator.in(state->source); },
                                                        state->out);
                                                if (source.empty()) {
                                                        return;
                                                }

                                                // this subscribe does not share the observer subscription
                                                // so that when it is unsubscribed the observer can be called
                                                // until the inner subscriptions have finished
                                                auto sink = make_subscriber<source_value_type>(
                                                        state->out,
                                                        state->sourceLifetime,
                                                        // on_next
                                                        [state](source_value_type st) {
                                                        if (state->collectionLifetime.is_subscribed()) {
                                                                state->selectedCollections.push_back(st);
                                                        }
                                                        else if (state->selectedCollections.empty()) {
                                                                state->subscribe_to(st);
                                                        }
                                                },
                                                        // on_error
                                                        [state](std::exception_ptr e) {
                                                        state->out.on_error(e);
                                                },
                                                        // on_completed
                                                        [state]() {
                                                        if (!state->collectionLifetime.is_subscribed() && state-
>selectedCollections.empty()) {
                                                                state->out.on_completed();
                                                        }
                                                }
                                                );
                                                auto selectedSink = on_exception(
                                                        [&](){return state->coordinator.out(sink); },
                                                        state->out);
                                                if (selectedSink.empty()) {
                                                        return;
                                                }
                                                source->subscribe(std::move(selectedSink.get()));
```

```
                                        }
                                private:
                                        concat_map& operator=(const concat_map&)RXCPP_DELETE;
                                };

                                template<class CollectionSelector, class ResultSelector, class Coordination>
                                class concat_map_factory
                                {
                                        typedef rxu::decay_t<CollectionSelector> collection_selector_type;
                                        typedef rxu::decay_t<ResultSelector> result_selector_type;
                                        typedef rxu::decay_t<Coordination> coordination_type;

                                        collection_selector_type selectorCollection;
                                        result_selector_type selectorResult;
                                        coordination_type coordination;
                                public:
                                        concat_map_factory(collection_selector_type s, result_selector_type rs, coordination_type sf)
                                                : selectorCollection(std::move(s))
                                                , selectorResult(std::move(rs))
                                                , coordination(std::move(sf))
                                        {
                                        }

                                        template<class Observable>
                                        auto operator()(Observable&& source)
                                                ->      observable<rxu::value_type_t<concat_map<Observable, CollectionSelector,
ResultSelector, Coordination>>, concat_map<Observable, CollectionSelector, ResultSelector, Coordination>> {
                                                return  observable<rxu::value_type_t<concat_map<Observable, CollectionSelector,
ResultSelector, Coordination>>, concat_map<Observable, CollectionSelector, ResultSelector, Coordination>>(
                                                                concat_map<Observable, CollectionSelector, ResultSelector,
Coordination>(std::forward<Observable>(source), selectorCollection, selectorResult, coordination));
                                        }
                                };

                        }

                        template<class CollectionSelector, class ResultSelector, class Coordination>
                        auto concat_map(CollectionSelector&& s, ResultSelector&& rs, Coordination&& sf)
                                ->      detail::concat_map_factory<CollectionSelector, ResultSelector, Coordination> {
                                return  detail::concat_map_factory<CollectionSelector, ResultSelector,
Coordination>(std::forward<CollectionSelector>(s), std::forward<ResultSelector>(rs), std::forward<Coordination>(sf));
                        }

                        template<class CollectionSelector, class Coordination, class CheckC = typename
std::enable_if<is_coordination<Coordination>::value>::type>
                        auto concat_map(CollectionSelector&& s, Coordination&& sf)
                                ->      detail::concat_map_factory<CollectionSelector, rxu::detail::take_at<1>, Coordination> {
                                return  detail::concat_map_factory<CollectionSelector, rxu::detail::take_at<1>,
Coordination>(std::forward<CollectionSelector>(s), rxu::take_at<1>(), std::forward<Coordination>(sf));
                        }

                        template<class CollectionSelector>
                        auto concat_map(CollectionSelector&& s)
                                ->      detail::concat_map_factory<CollectionSelector, rxu::detail::take_at<1>, identity_one_worker> {
                                return  detail::concat_map_factory<CollectionSelector, rxu::detail::take_at<1>,
identity_one_worker>(std::forward<CollectionSelector>(s), rxu::take_at<1>(), identity_current_thread());
                        }


                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_CONNECT_FOREVER_HPP)
#define RXCPP_OPERATORS_RX_CONNECT_FOREVER_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class ConnectableObservable>
                        struct connect_forever : public operator_base<T>
                        {
                                typedef rxu::decay_t<ConnectableObservable> source_type;
```

```
                                        source_type source;

                                        explicit connect_forever(source_type o)
                                                : source(std::move(o))
                                        {
                                                source.connect();
                                        }

                                        template<class Subscriber>
                                        void on_subscribe(Subscriber&& o) const {
                                                source.subscribe(std::forward<Subscriber>(o));
                                        }
                                };

                                class connect_forever_factory
                                {
                                public:
                                        connect_forever_factory() {}
                                        template<class... TN>
                                        auto operator()(connectable_observable<TN...>&& source)
                                                ->      observable<rxu::value_type_t<connectable_observable<TN...>>,
connect_forever<rxu::value_type_t<connectable_observable<TN...>>, connectable_observable<TN...>>> {
                                                return  observable<rxu::value_type_t<connectable_observable<TN...>>,
connect_forever<rxu::value_type_t<connectable_observable<TN...>>, connectable_observable<TN...>>>(
                                                        connect_forever<rxu::value_type_t<connectable_observable<TN...>>,
connectable_observable<TN...>>(std::move(source)));
                                        }
                                        template<class... TN>
                                        auto operator()(const connectable_observable<TN...>& source)
                                                ->      observable<rxu::value_type_t<connectable_observable<TN...>>,
connect_forever<rxu::value_type_t<connectable_observable<TN...>>, connectable_observable<TN...>>> {
                                                return  observable<rxu::value_type_t<connectable_observable<TN...>>,
connect_forever<rxu::value_type_t<connectable_observable<TN...>>, connectable_observable<TN...>>>(
                                                        connect_forever<rxu::value_type_t<connectable_observable<TN...>>,
connectable_observable<TN...>>(source));
                                        }
                                };

                        }

                        inline auto connect_forever()
                                ->      detail::connect_forever_factory {
                                return  detail::connect_forever_factory();
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_FINALLY_HPP)
#define RXCPP_OPERATORS_RX_FINALLY_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class LastCall>
                        struct finally
                        {
                                typedef rxu::decay_t<T> source_value_type;
                                typedef rxu::decay_t<LastCall> last_call_type;
                                last_call_type last_call;

                                finally(last_call_type lc)
                                        : last_call(std::move(lc))
                                {
                                }

                                template<class Subscriber>
                                struct finally_observer
                                {
                                        typedef finally_observer<Subscriber> this_type;
                                        typedef source_value_type value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<value_type, this_type> observer_type;
```

```
                                                dest_type dest;

                                                finally_observer(dest_type d)
                                                        : dest(std::move(d))
                                                {
                                                }
                                                void on_next(source_value_type v) const {
                                                        dest.on_next(v);
                                                }
                                                void on_error(std::exception_ptr e) const {
                                                        dest.on_error(e);
                                                }
                                                void on_completed() const {
                                                        dest.on_completed();
                                                }

                                                static subscriber<value_type, observer<value_type, this_type>> make(dest_type d, const
last_call_type& lc) {
                                                        auto dl = d.get_subscription();
                                                        composite_subscription cs;
                                                        dl.add(cs);
                                                        cs.add([=](){
                                                                dl.unsubscribe();
                                                                lc();
                                                        });
                                                        return make_subscriber<value_type>(cs, this_type(d));
                                                }
                                        };

                                        template<class Subscriber>
                                        auto operator()(Subscriber dest) const
                                                -> decltype(finally_observer<Subscriber>::make(std::move(dest), last_call)) {
                                                return     finally_observer<Subscriber>::make(std::move(dest), last_call);
                                        }
                                };

                                template<class LastCall>
                                class finally_factory
                                {
                                        typedef rxu::decay_t<LastCall> last_call_type;
                                        last_call_type last_call;
                                public:
                                        finally_factory(last_call_type lc) : last_call(std::move(lc)) {}
                                        template<class Observable>
                                        auto operator()(Observable&& source)
                                                -> decltype(source.template
lift<rxu::value_type_t<rxu::decay_t<Observable>>>(finally<rxu::value_type_t<rxu::decay_t<Observable>>, last_call_type>(last_call))) {
                                                return     source.template
lift<rxu::value_type_t<rxu::decay_t<Observable>>>(finally<rxu::value_type_t<rxu::decay_t<Observable>>, last_call_type>(last_call));
                                        }
                                };

                        }

                        template<class LastCall>
                        auto finally(LastCall lc)
                                ->      detail::finally_factory<LastCall> {
                                return  detail::finally_factory<LastCall>(std::move(lc));
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_FLATMAP_HPP)
#define RXCPP_OPERATORS_RX_FLATMAP_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class Observable, class CollectionSelector, class ResultSelector, class Coordination>
                        struct flat_map_traits {
                                typedef rxu::decay_t<Observable> source_type;
                                typedef rxu::decay_t<CollectionSelector> collection_selector_type;
```

```
                                        typedef rxu::decay_t<ResultSelector> result_selector_type;
                                        typedef rxu::decay_t<Coordination> coordination_type;

                                        typedef typename source_type::value_type source_value_type;

                                        struct tag_not_valid {};
                                        template<class CV, class CCS>
                                        static auto collection_check(int) -> decltype((*(CCS*)nullptr)(*(CV*)nullptr));
                                        template<class CV, class CCS>
                                        static tag_not_valid collection_check(...);

                                        static_assert(!std::is_same<decltype(collection_check<source_value_type,
collection_selector_type>(0)), tag_not_valid>::value, "flat_map CollectionSelector must be a function with the signature
observable(flat_map::source_value_type)");

                                        typedef
rxu::decay_t<decltype((*(collection_selector_type*)nullptr)((*(source_value_type*)nullptr)))> collection_type;

                                        static_assert(is_observable<collection_type>::value, "flat_map CollectionSelector must return an
observable");

                                        typedef typename collection_type::value_type collection_value_type;

                                        template<class CV, class CCV, class CRS>
                                        static auto result_check(int) -> decltype((*(CRS*)nullptr)(*(CV*)nullptr, *(CCV*)nullptr));
                                        template<class CV, class CCV, class CRS>
                                        static tag_not_valid result_check(...);

                                        static_assert(!std::is_same<decltype(result_check<source_value_type, collection_value_type,
result_selector_type>(0)), tag_not_valid>::value, "flat_map ResultSelector must be a function with the signature
flat_map::value_type(flat_map::source_value_type, flat_map::collection_value_type)");

                                        typedef rxu::decay_t<decltype((*(result_selector_type*)nullptr)(*(source_value_type*)nullptr,
*(collection_value_type*)nullptr))> value_type;
                                };

                                template<class Observable, class CollectionSelector, class ResultSelector, class Coordination>
                                struct flat_map
                                        : public operator_base<rxu::value_type_t<flat_map_traits<Observable, CollectionSelector,
ResultSelector, Coordination>>>
                                {
                                        typedef flat_map<Observable, CollectionSelector, ResultSelector, Coordination> this_type;
                                        typedef flat_map_traits<Observable, CollectionSelector, ResultSelector, Coordination> traits;

                                        typedef typename traits::source_type source_type;
                                        typedef typename traits::collection_selector_type collection_selector_type;
                                        typedef typename traits::result_selector_type result_selector_type;

                                        typedef typename traits::source_value_type source_value_type;
                                        typedef typename traits::collection_type collection_type;
                                        typedef typename traits::collection_value_type collection_value_type;

                                        typedef typename traits::coordination_type coordination_type;
                                        typedef typename coordination_type::coordinator_type coordinator_type;

                                        struct values
                                        {
                                                values(source_type o, collection_selector_type s, result_selector_type rs, coordination_type
sf)
                                                : source(std::move(o))
                                                , selectCollection(std::move(s))
                                                , selectResult(std::move(rs))
                                                , coordination(std::move(sf))
                                                {
                                                }
                                                source_type source;
                                                collection_selector_type selectCollection;
                                                result_selector_type selectResult;
                                                coordination_type coordination;
                                        };
                                        values initial;

                                        flat_map(source_type o, collection_selector_type s, result_selector_type rs, coordination_type sf)
                                                : initial(std::move(o), std::move(s), std::move(rs), std::move(sf))
                                        {
                                        }

                                        template<class Subscriber>
                                        void on_subscribe(Subscriber scbr) const {
                                                static_assert(is_subscriber<Subscriber>::value, "subscribe must be passed a subscriber");

                                                typedef Subscriber output_type;
```

```
                                        struct state_type
                                                : public std::enable_shared_from_this<state_type>
                                                , public values
                                        {
                                                state_type(values i, coordinator_type coor, output_type oarg)
                                                : values(std::move(i))
                                                , pendingCompletions(0)
                                                , coordinator(std::move(coor))
                                                , out(std::move(oarg))
                                                {
                                                }
                                                // on_completed on the output must wait until all the
                                                // subscriptions have received on_completed
                                                int pendingCompletions;
                                                coordinator_type coordinator;
                                                output_type out;
                                        };

                                        auto coordinator = initial.coordination.create_coordinator(scbr.get_subscription());

                                        // take a copy of the values for each subscription
                                        auto state = std::make_shared<state_type>(initial, std::move(coordinator),
std::move(scbr));


                                        composite_subscription outercs;

                                        // when the out observer is unsubscribed all the
                                        // inner subscriptions are unsubscribed as well
                                        state->out.add(outercs);

                                        auto source = on_exception(
                                                [&](){return state->coordinator.in(state->source); },
                                                state->out);
                                        if (source.empty()) {
                                                return;
                                        }

                                        ++state->pendingCompletions;
                                        // this subscribe does not share the observer subscription
                                        // so that when it is unsubscribed the observer can be called
                                        // until the inner subscriptions have finished
                                        auto sink = make_subscriber<source_value_type>(
                                                state->out,
                                                outercs,
                                                // on_next
                                                [state](source_value_type st) {

                                                composite_subscription innercs;

                                                // when the out observer is unsubscribed all the
                                                // inner subscriptions are unsubscribed as well
                                                auto innercstoken = state->out.add(innercs);

                                                innercs.add(make_subscription([state, innercstoken](){
                                                        state->out.remove(innercstoken);
                                                }));

                                                auto selectedCollection = state->selectCollection(st);
                                                auto selectedSource = state->coordinator.in(selectedCollection);

                                                ++state->pendingCompletions;
                                                // this subscribe does not share the source subscription
                                                // so that when it is unsubscribed the source will continue
                                                auto sinkInner = make_subscriber<collection_value_type>(
                                                        state->out,
                                                        innercs,
                                                        // on_next
                                                        [state, st](collection_value_type ct) {
                                                        auto selectedResult = state->selectResult(st, std::move(ct));
                                                        state->out.on_next(std::move(selectedResult));
                                                },
                                                        // on_error
                                                        [state](std::exception_ptr e) {
                                                        state->out.on_error(e);
                                                },
                                                        //on_completed
                                                        [state](){
                                                        if (--state->pendingCompletions == 0) {
                                                                state->out.on_completed();
                                                        }
                                                }
```

```
                                                );

                                                auto selectedSinkInner = state->coordinator.out(sinkInner);
                                                selectedSource.subscribe(std::move(selectedSinkInner));
                                        },
                                                // on_error
                                                [state](std::exception_ptr e) {
                                                state->out.on_error(e);
                                        },
                                                // on_completed
                                                [state]() {
                                                if (--state->pendingCompletions == 0) {
                                                        state->out.on_completed();
                                                }
                                        }
                                        );

                                        auto selectedSink = on_exception(
                                                [&](){return state->coordinator.out(sink); },
                                                state->out);
                                        if (selectedSink.empty()) {
                                                return;
                                        }

                                        source->subscribe(std::move(selectedSink.get()));

                                }
                        };

                        template<class CollectionSelector, class ResultSelector, class Coordination>
                        class flat_map_factory
                        {
                                typedef rxu::decay_t<CollectionSelector> collection_selector_type;
                                typedef rxu::decay_t<ResultSelector> result_selector_type;
                                typedef rxu::decay_t<Coordination> coordination_type;

                                collection_selector_type selectorCollection;
                                result_selector_type selectorResult;
                                coordination_type coordination;
                        public:
                                flat_map_factory(collection_selector_type s, result_selector_type rs, coordination_type sf)
                                        : selectorCollection(std::move(s))
                                        , selectorResult(std::move(rs))
                                        , coordination(std::move(sf))
                                {
                                }

                                template<class Observable>
                                auto operator()(Observable&& source)
                                        ->      observable<rxu::value_type_t<flat_map<Observable, CollectionSelector,
ResultSelector, Coordination>>, flat_map<Observable, CollectionSelector, ResultSelector, Coordination>> {
                                        return  observable<rxu::value_type_t<flat_map<Observable, CollectionSelector,
ResultSelector, Coordination>>, flat_map<Observable, CollectionSelector, ResultSelector, Coordination>>(
                                                flat_map<Observable, CollectionSelector, ResultSelector,
Coordination>(std::forward<Observable>(source), selectorCollection, selectorResult, coordination));
                                }
                        };

                }

                template<class CollectionSelector, class ResultSelector, class Coordination>
                auto flat_map(CollectionSelector&& s, ResultSelector&& rs, Coordination&& sf)
                        ->      detail::flat_map_factory<CollectionSelector, ResultSelector, Coordination> {
                        return  detail::flat_map_factory<CollectionSelector, ResultSelector,
Coordination>(std::forward<CollectionSelector>(s), std::forward<ResultSelector>(rs), std::forward<Coordination>(sf));
                }

                template<class CollectionSelector, class Coordination, class CheckC = typename
std::enable_if<is_coordination<Coordination>::value>::type>
                auto flat_map(CollectionSelector&& s, Coordination&& sf)
                        ->      detail::flat_map_factory<CollectionSelector, rxu::detail::take_at<1>, Coordination> {
                        return  detail::flat_map_factory<CollectionSelector, rxu::detail::take_at<1>,
Coordination>(std::forward<CollectionSelector>(s), rxu::take_at<1>(), std::forward<Coordination>(sf));
                }

                template<class CollectionSelector>
                auto flat_map(CollectionSelector&& s)
                        ->      detail::flat_map_factory<CollectionSelector, rxu::detail::take_at<1>, identity_one_worker> {
                        return  detail::flat_map_factory<CollectionSelector, rxu::detail::take_at<1>,
identity_one_worker>(std::forward<CollectionSelector>(s), rxu::take_at<1>(), identity_current_thread());
                }
```

```
		}

}

#endif


#if !defined(RXCPP_OPERATORS_RX_LIFT_HPP)
#define RXCPP_OPERATORS_RX_LIFT_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

		namespace detail {

				template<class V, class S, class F>
				struct is_lift_function_for {

						struct tag_not_valid {};
						template<class CS, class CF>
						static auto check(int) -> decltype((*(CF*)nullptr)(*(CS*)nullptr));
						template<class CS, class CF>
						static tag_not_valid check(...);

						using for_type = rxu::decay_t<S>;
						using func_type = rxu::decay_t<F>;
						using detail_result = decltype(check<for_type, func_type>(0));

						static const bool value = rxu::all_true_type<
								is_subscriber<detail_result>,
								is_subscriber<for_type>,
								std::is_convertible<V, typename rxu::value_type_from<detail_result>::type >> ::value;
				};

		}

		namespace operators {

				namespace detail {

						template<class ResultType, class SourceOperator, class Operator>
						struct lift_traits
						{
								typedef rxu::decay_t<ResultType> result_value_type;
								typedef rxu::decay_t<SourceOperator> source_operator_type;
								typedef rxu::decay_t<Operator> operator_type;

								typedef typename source_operator_type::value_type source_value_type;
						};

						template<class ResultType, class SourceOperator, class Operator>
						struct lift_operator : public operator_base<typename lift_traits<ResultType, SourceOperator,
Operator>::result_value_type>
						{
								typedef lift_traits<ResultType, SourceOperator, Operator> traits;
								typedef typename traits::source_operator_type source_operator_type;
								typedef typename traits::operator_type operator_type;
								source_operator_type source;
								operator_type chain;

								lift_operator(source_operator_type s, operator_type op)
										: source(std::move(s))
										, chain(std::move(op))
								{
								}
								template<class Subscriber>
								void on_subscribe(Subscriber o) const {
										auto lifted = chain(std::move(o));
										trace_activity().lift_enter(source, chain, o, lifted);
										source.on_subscribe(std::move(lifted));
										trace_activity().lift_return(source, chain);
								}
						};

						template<class ResultType, class Operator>
						class lift_factory
						{
								typedef rxu::decay_t<Operator> operator_type;
								operator_type chain;
						public:
								lift_factory(operator_type op) : chain(std::move(op)) {}
```

```
                                        template<class Observable>
                                        auto operator()(const Observable& source)
                                                -> decltype(source.template lift<ResultType>(chain)) {
                                                return      source.template lift<ResultType>(chain);
                                                static_assert(rxcpp::detail::is_lift_function_for<rxu::value_type_t<Observable>,
subscriber<ResultType>, Operator>::value, "Function passed for lift() must have the signature subscriber<...>(subscriber<T, ...>)");
                                        }
                                };

                        }

                        template<class ResultType, class Operator>
                        auto lift(Operator&& op)
                                ->      detail::lift_factory<ResultType, Operator> {
                                return  detail::lift_factory<ResultType, Operator>(std::forward<Operator>(op));
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_MAP_HPP)
#define RXCPP_OPERATORS_RX_MAP_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {


                        template<class T, class Selector>
                        struct map
                        {
                                typedef rxu::decay_t<T> source_value_type;
                                typedef rxu::decay_t<Selector> select_type;
                                typedef decltype((*(select_type*)nullptr)(*(source_value_type*)nullptr)) value_type;
                                select_type selector;

                                map(select_type s)
                                        : selector(std::move(s))
                                {
                                }

                                template<class Subscriber>
                                struct map_observer
                                {
                                        typedef map_observer<Subscriber> this_type;
                                        typedef decltype((*(select_type*)nullptr)(*(source_value_type*)nullptr)) value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<T, this_type> observer_type;
                                        dest_type dest;
                                        mutable select_type selector;

                                        map_observer(dest_type d, select_type s)
                                                : dest(std::move(d))
                                                , selector(std::move(s))
                                        {
                                        }
                                        template<class Value>
                                        void on_next(Value&& v) const {
                                                auto selected = on_exception(
                                                        [&](){
                                                        return this->selector(std::forward<Value>(v)); },
                                                                dest);
                                                        if (selected.empty()) {
                                                                return;
                                                        }
                                                        dest.on_next(std::move(selected.get()));
                                        }
                                        void on_error(std::exception_ptr e) const {
                                                dest.on_error(e);
                                        }
                                        void on_completed() const {
                                                dest.on_completed();
                                        }
```

```
                                                static subscriber<T, observer_type> make(dest_type d, select_type s) {
                                                        auto cs = d.get_subscription();
                                                        return make_subscriber<T>(std::move(cs),
observer_type(this_type(std::move(d), std::move(s))));
                                                }
                                        };

                                        template<class Subscriber>
                                        auto operator()(Subscriber dest) const
                                                -> decltype(map_observer<Subscriber>::make(std::move(dest), selector)) {
                                                return      map_observer<Subscriber>::make(std::move(dest), selector);
                                        }
                                };

                                template<class Selector>
                                class map_factory
                                {
                                        typedef rxu::decay_t<Selector> select_type;
                                        select_type selector;
                                public:
                                        map_factory(select_type s) : selector(std::move(s)) {}
                                        template<class Observable>
                                        auto operator()(Observable&& source)
                                                -> decltype(source.template
lift<rxu::value_type_t<map<rxu::value_type_t<rxu::decay_t<Observable>>,
select_type>>>(map<rxu::value_type_t<rxu::decay_t<Observable>>, select_type>(selector))) {
                                                return      source.template
lift<rxu::value_type_t<map<rxu::value_type_t<rxu::decay_t<Observable>>,
select_type>>>(map<rxu::value_type_t<rxu::decay_t<Observable>>, select_type>(selector));
                                        }
                                };

                        }

                        template<class Selector>
                        auto map(Selector&& p)
                                ->      detail::map_factory<Selector> {
                                return  detail::map_factory<Selector>(std::forward<Selector>(p));
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_MERGE_HPP)
#define RXCPP_OPERATORS_RX_MERGE_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Observable, class Coordination>
                        struct merge
                                : public operator_base<rxu::value_type_t<rxu::decay_t<T>>>
                        {
                                //static_assert(is_observable<Observable>::value, "merge requires an observable");
                                //static_assert(is_observable<T>::value, "merge requires an observable that contains observables");

                                typedef merge<T, Observable, Coordination> this_type;

                                typedef rxu::decay_t<T> source_value_type;
                                typedef rxu::decay_t<Observable> source_type;

                                typedef typename source_type::source_operator_type source_operator_type;
                                typedef typename source_value_type::value_type value_type;

                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef typename coordination_type::coordinator_type coordinator_type;

                                struct values
                                {
                                        values(source_operator_type o, coordination_type sf)
                                        : source_operator(std::move(o))
                                        , coordination(std::move(sf))
                                        {
```

```
                }
                source_operator_type source_operator;
                coordination_type coordination;
        };
        values initial;

        merge(const source_type& o, coordination_type sf)
                : initial(o.source_operator, std::move(sf))
        {
        }

        template<class Subscriber>
        void on_subscribe(Subscriber scbr) const {
                static_assert(is_subscriber<Subscriber>::value, "subscribe must be passed a subscriber");

                typedef Subscriber output_type;

                struct merge_state_type
                        : public std::enable_shared_from_this<merge_state_type>
                        , public values
                {
                        merge_state_type(values i, coordinator_type coor, output_type oarg)
                        : values(i)
                        , source(i.source_operator)
                        , pendingCompletions(0)
                        , coordinator(std::move(coor))
                        , out(std::move(oarg))
                        {
                        }
                        observable<source_value_type, source_operator_type> source;
                        // on_completed on the output must wait until all the
                        // subscriptions have received on_completed
                        int pendingCompletions;
                        coordinator_type coordinator;
                        output_type out;
                };

                auto coordinator = initial.coordination.create_coordinator(scbr.get_subscription());

                // take a copy of the values for each subscription
                auto state = std::make_shared<merge_state_type>(initial, std::move(coordinator),
std::move(scbr));

                composite_subscription outercs;

                // when the out observer is unsubscribed all the
                // inner subscriptions are unsubscribed as well
                state->out.add(outercs);

                auto source = on_exception(
                        [&](){return state->coordinator.in(state->source); },
                        state->out);
                if (source.empty()) {
                        return;
                }

                ++state->pendingCompletions;
                // this subscribe does not share the observer subscription
                // so that when it is unsubscribed the observer can be called
                // until the inner subscriptions have finished
                auto sink = make_subscriber<source_value_type>(
                        state->out,
                        outercs,
                        // on_next
                        [state](source_value_type st) {

                        composite_subscription innercs;

                        // when the out observer is unsubscribed all the
                        // inner subscriptions are unsubscribed as well
                        auto innercstoken = state->out.add(innercs);

                        innercs.add(make_subscription([state, innercstoken](){
                                state->out.remove(innercstoken);
                        }));

                        auto selectedSource = state->coordinator.in(st);

                        ++state->pendingCompletions;
                        // this subscribe does not share the source subscription
                        // so that when it is unsubscribed the source will continue
                        auto sinkInner = make_subscriber<value_type>(
```

```
                                                                state->out,
                                                                innercs,
                                                                // on_next
                                                                [state, st](value_type ct) {
                                                                state->out.on_next(std::move(ct));
                                                        },
                                                                // on_error
                                                                [state](std::exception_ptr e) {
                                                                state->out.on_error(e);
                                                        },
                                                                //on_completed
                                                                [state](){
                                                                if (--state->pendingCompletions == 0) {
                                                                        state->out.on_completed();
                                                                }
                                                        }
                                                        );

                                                        auto selectedSinkInner = state->coordinator.out(sinkInner);
                                                        selectedSource.subscribe(std::move(selectedSinkInner));
                                                },
                                                        // on_error
                                                        [state](std::exception_ptr e) {
                                                        state->out.on_error(e);
                                                },
                                                        // on_completed
                                                        [state]() {
                                                        if (--state->pendingCompletions == 0) {
                                                                state->out.on_completed();
                                                        }
                                                }
                                                );
                                                auto selectedSink = on_exception(
                                                        [&](){return state->coordinator.out(sink); },
                                                        state->out);
                                                if (selectedSink.empty()) {
                                                        return;
                                                }
                                                source->subscribe(std::move(selectedSink.get()));
                                        }
                                };

                                template<class Coordination>
                                class merge_factory
                                {
                                        typedef rxu::decay_t<Coordination> coordination_type;

                                        coordination_type coordination;
                                public:
                                        merge_factory(coordination_type sf)
                                                : coordination(std::move(sf))
                                        {
                                        }

                                        template<class Observable>
                                        auto operator()(Observable source)
                                                ->      observable<rxu::value_type_t<merge<rxu::value_type_t<Observable>, Observable,
Coordination>>, merge<rxu::value_type_t<Observable>, Observable, Coordination>> {
                                                return  observable<rxu::value_type_t<merge<rxu::value_type_t<Observable>, Observable,
Coordination>>, merge<rxu::value_type_t<Observable>, Observable, Coordination>>(
                                                        merge<rxu::value_type_t<Observable>, Observable,
Coordination>(std::move(source), coordination));
                                        }
                                };

                        }

                        inline auto merge()
                                ->      detail::merge_factory<identity_one_worker> {
                                return  detail::merge_factory<identity_one_worker>(identity_current_thread());
                        }

                        template<class Coordination>
                        auto merge(Coordination&& sf)
                                ->      detail::merge_factory<Coordination> {
                                return  detail::merge_factory<Coordination>(std::forward<Coordination>(sf));
                        }

                }

}
```

```
#endif


#if !defined(RXCPP_OPERATORS_RX_MULTICAST_HPP)
#define RXCPP_OPERATORS_RX_MULTICAST_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace operators {

		namespace detail {

			template<class T, class Observable, class Subject>
			struct multicast : public operator_base<T>
			{
				typedef rxu::decay_t<Observable> source_type;
				typedef rxu::decay_t<Subject> subject_type;

				struct multicast_state : public std::enable_shared_from_this<multicast_state>
				{
					multicast_state(source_type o, subject_type sub)
					: source(std::move(o))
					, subject_value(std::move(sub))
					{
					}
					source_type source;
					subject_type subject_value;
					rxu::detail::maybe<typename composite_subscription::weak_subscription> connection;
				};

				std::shared_ptr<multicast_state> state;

				multicast(source_type o, subject_type sub)
					: state(std::make_shared<multicast_state>(std::move(o), std::move(sub)))
				{
				}
				template<class Subscriber>
				void on_subscribe(Subscriber&& o) const {
					state->subject_value.get_observable().subscribe(std::forward<Subscriber>(o));
				}
				void on_connect(composite_subscription cs) const {
					if (state->connection.empty()) {
						auto destination = state->subject_value.get_subscriber();

						// the lifetime of each connect is nested in the subject lifetime
						state->connection.reset(destination.add(cs));

						auto localState = state;

						// when the connection is finished it should shutdown the connection
						cs.add(
							[destination, localState](){
							if (!localState->connection.empty()) {
								destination.remove(localState->connection.get());
								localState->connection.reset();
							}
						});

						// use cs not destination for lifetime of subscribe.
						state->source.subscribe(cs, destination);
					}
				}
			};

			template<class Subject>
			class multicast_factory
			{
				Subject caster;
			public:
				multicast_factory(Subject sub)
					: caster(std::move(sub))
				{
				}
				template<class Observable>
				auto operator()(Observable&& source)
					-> connectable_observable<rxu::value_type_t<rxu::decay_t<Observable>>,
multicast<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, Subject>> {
					return connectable_observable<rxu::value_type_t<rxu::decay_t<Observable>>,
multicast<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, Subject>>(
```

```
                                                multicast<rxu::value_type_t<rxu::decay_t<Observable>>, Observable,
Subject>(std::forward<Observable>(source), caster));
                                        }
                                };

                        }

                        template<class Subject>
                        inline auto multicast(Subject sub)
                                ->      detail::multicast_factory<Subject> {
                                return  detail::multicast_factory<Subject>(std::move(sub));
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_OBSERVE_ON_HPP)
#define RXCPP_OPERATORS_RX_OBSERVE_ON_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Coordination>
                        struct observe_on
                        {
                                typedef rxu::decay_t<T> source_value_type;

                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef typename coordination_type::coordinator_type coordinator_type;

                                coordination_type coordination;

                                observe_on(coordination_type cn)
                                        : coordination(std::move(cn))
                                {
                                }

                                template<class Subscriber>
                                struct observe_on_observer
                                {
                                        typedef observe_on_observer<Subscriber> this_type;
                                        typedef source_value_type value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<value_type, this_type> observer_type;

                                        typedef rxn::notification<T> notification_type;
                                        typedef typename notification_type::type base_notification_type;
                                        typedef std::deque<base_notification_type> queue_type;

                                        struct mode
                                        {
                                                enum type {
                                                        Invalid = 0,
                                                        Processing,
                                                        Empty,
                                                        Disposed,
                                                        Errored
                                                };
                                        };
                                        struct observe_on_state : std::enable_shared_from_this<observe_on_state>
                                        {
                                                mutable std::mutex lock;
                                                mutable queue_type fill_queue;
                                                mutable queue_type drain_queue;
                                                composite_subscription lifetime;
                                                mutable typename mode::type current;
                                                coordinator_type coordinator;
                                                dest_type destination;

                                                observe_on_state(dest_type d, coordinator_type coor, composite_subscription
cs)
                                                        : lifetime(std::move(cs))
                                                        , current(mode::Empty)
```

```
                , coordinator(std::move(coor))
                , destination(std::move(d))
        {
        }

        void finish(std::unique_lock<std::mutex>& guard, typename mode::type end)
const {
                if (!guard.owns_lock()) {
                        std::terminate();
                }
                if (current == mode::Errored || current == mode::Disposed) { return;
}
                current = end;
                queue_type fill_expired;
                swap(fill_expired, fill_queue);
                queue_type drain_expired;
                swap(drain_expired, drain_queue);
                RXCPP_UNWIND_AUTO([&](){guard.lock(); });
                guard.unlock();
                lifetime.unsubscribe();
                destination.unsubscribe();
        }

        void ensure_processing(std::unique_lock<std::mutex>& guard) const {
                if (!guard.owns_lock()) {
                        std::terminate();
                }
                if (current == mode::Empty) {
                        current = mode::Processing;

                        if (!lifetime.is_subscribed() && fill_queue.empty() &&
drain_queue.empty()) {
                                finish(guard, mode::Disposed);
                        }

                        auto keepAlive = this->shared_from_this();

                        auto drain = [keepAlive, this](const rxsc::schedulable&
self){
                                using std::swap;
                                try {
                                        for (;;) {
                                                if (drain_queue.empty()
|| !destination.is_subscribed()) {
        std::unique_lock<std::mutex> guard(lock);
                                                        if
(!destination.is_subscribed() ||
        (!lifetime.is_subscribed() && fill_queue.empty() && drain_queue.empty())) {
        finish(guard, mode::Disposed);
        return;
                                                        }
                                                        if
(drain_queue.empty()) {
                                                                if
(fill_queue.empty()) {
        current = mode::Empty;
        return;
                                                                }
        swap(fill_queue, drain_queue);
                                                        }
                                                }
                                                auto notification =
std::move(drain_queue.front());
                                                drain_queue.pop_front();
                                                notification-
>accept(destination);
        std::unique_lock<std::mutex> guard(lock);
                                                self();
                                                if
(lifetime.is_subscribed()) break;
                                        }
                                }
                                catch (...) {
```

```
                destination.on_error(std::current_exception());
                                                                                std::unique_lock<std::mutex>
guard(lock);
                                                                                finish(guard, mode::Errored);
                                                                        }
                                                                };

                                                                auto selectedDrain = on_exception(
                                                                        [&](){return coordinator.act(drain); },
                                                                        destination);
                                                                if (selectedDrain.empty()) {
                                                                        finish(guard, mode::Errored);
                                                                        return;
                                                                }

                                                                auto processor = coordinator.get_worker();

                                                                RXCPP_UNWIND_AUTO([&](){guard.lock(); });
                                                                guard.unlock();

                                                                processor.schedule(selectedDrain.get());
                                                        }
                                                }
                                        };
                                        std::shared_ptr<observe_on_state> state;

                                        observe_on_observer(dest_type d, coordinator_type coor, composite_subscription cs)
                                                : state(std::make_shared<observe_on_state>(std::move(d), std::move(coor),
std::move(cs)))
                                        {
                                        }

                                        void on_next(source_value_type v) const {
                                                std::unique_lock<std::mutex> guard(state->lock);
                                                if (state->current == mode::Errored || state->current == mode::Disposed) {
return; }
                                                state->fill_queue.push_back(notification_type::on_next(std::move(v)));
                                                state->ensure_processing(guard);
                                        }
                                        void on_error(std::exception_ptr e) const {
                                                std::unique_lock<std::mutex> guard(state->lock);
                                                if (state->current == mode::Errored || state->current == mode::Disposed) {
return; }
                                                state->fill_queue.push_back(notification_type::on_error(e));
                                                state->ensure_processing(guard);
                                        }
                                        void on_completed() const {
                                                std::unique_lock<std::mutex> guard(state->lock);
                                                if (state->current == mode::Errored || state->current == mode::Disposed) {
return; }
                                                state->fill_queue.push_back(notification_type::on_completed());
                                                state->ensure_processing(guard);
                                        }

                                        static subscriber<value_type, observer<value_type, this_type>> make(dest_type d,
coordination_type cn, composite_subscription cs = composite_subscription()) {
                                                auto coor = cn.create_coordinator(d.get_subscription());
                                                d.add(cs);

                                                this_type o(d, std::move(coor), cs);
                                                auto keepAlive = o.state;
                                                cs.add([=](){
                                                        std::unique_lock<std::mutex> guard(keepAlive->lock);
                                                        keepAlive->ensure_processing(guard);
                                                });

                                                return make_subscriber<value_type>(d, cs,
make_observer<value_type>(std::move(o)));
                                        }
                                };

                                template<class Subscriber>
                                auto operator()(Subscriber dest) const
                                        -> decltype(observe_on_observer<decltype(dest.as_dynamic())>::make(dest.as_dynamic(),
coordination)) {
                                        return     observe_on_observer<decltype(dest.as_dynamic())>::make(dest.as_dynamic(),
coordination);
                                }
                        };

                        template<class Coordination>
```

```
                                class observe_on_factory
                                {
                                        typedef rxu::decay_t<Coordination> coordination_type;
                                        coordination_type coordination;
                                public:
                                        observe_on_factory(coordination_type cn) : coordination(std::move(cn)) {}
                                        template<class Observable>
                                        auto operator()(Observable&& source)
                                                -> decltype(source.template
lift<rxu::value_type_t<rxu::decay_t<Observable>>>(observe_on<rxu::value_type_t<rxu::decay_t<Observable>>,
coordination_type>(coordination))) {
                                                return      source.template
lift<rxu::value_type_t<rxu::decay_t<Observable>>>(observe_on<rxu::value_type_t<rxu::decay_t<Observable>>,
coordination_type>(coordination));
                                        }
                                };

                        }

                        template<class Coordination>
                        auto observe_on(Coordination cn)
                                ->      detail::observe_on_factory<Coordination> {
                                return  detail::observe_on_factory<Coordination>(std::move(cn));
                        }


        }

        class observe_on_one_worker : public coordination_base
        {
                rxsc::scheduler factory;

                class input_type
                {
                        rxsc::worker controller;
                        rxsc::scheduler factory;
                        identity_one_worker coordination;
                public:
                        explicit input_type(rxsc::worker w)
                                : controller(w)
                                , factory(rxsc::make_same_worker(w))
                                , coordination(factory)
                        {
                        }
                        inline rxsc::worker get_worker() const {
                                return controller;
                        }
                        inline rxsc::scheduler get_scheduler() const {
                                return factory;
                        }
                        inline rxsc::scheduler::clock_type::time_point now() const {
                                return factory.now();
                        }
                        template<class Observable>
                        auto in(Observable o) const
                                -> decltype(o.observe_on(coordination)) {
                                return      o.observe_on(coordination);
                        }
                        template<class Subscriber>
                        auto out(Subscriber s) const
                                -> Subscriber {
                                return s;
                        }
                        template<class F>
                        auto act(F f) const
                                -> F {
                                return f;
                        }
                };

        public:

                explicit observe_on_one_worker(rxsc::scheduler sc) : factory(sc) {}

                typedef coordinator<input_type> coordinator_type;

                inline rxsc::scheduler::clock_type::time_point now() const {
                        return factory.now();
                }

                inline coordinator_type create_coordinator(composite_subscription cs = composite_subscription()) const {
                        auto w = factory.create_worker(std::move(cs));
```

```
                return coordinator_type(input_type(std::move(w)));
            }
        };

        inline observe_on_one_worker observe_on_run_loop(const rxsc::run_loop& rl) {
            static observe_on_one_worker r(rxsc::make_run_loop(rl));
            return r;
        }

        inline observe_on_one_worker observe_on_event_loop() {
            static observe_on_one_worker r(rxsc::make_event_loop());
            return r;
        }

        inline observe_on_one_worker observe_on_new_thread() {
            static observe_on_one_worker r(rxsc::make_new_thread());
            return r;
        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_ON_ERROR_RESUME_NEXT_HPP)
#define RXCPP_OPERATORS_RX_ON_ERROR_RESUME_NEXT_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

    namespace operators {

        namespace detail {


            template<class T, class Selector>
            struct on_error_resume_next
            {
                typedef rxu::decay_t<T> value_type;
                typedef rxu::decay_t<Selector> select_type;
                typedef decltype((*(select_type*)nullptr)(std::exception_ptr())) fallback_type;
                select_type selector;

                on_error_resume_next(select_type s)
                    : selector(std::move(s))
                {
                }

                template<class Subscriber>
                struct on_error_resume_next_observer
                {
                    typedef on_error_resume_next_observer<Subscriber> this_type;
                    typedef rxu::decay_t<T> value_type;
                    typedef rxu::decay_t<Selector> select_type;
                    typedef decltype((*(select_type*)nullptr)(std::exception_ptr())) fallback_type;
                    typedef rxu::decay_t<Subscriber> dest_type;
                    typedef observer<T, this_type> observer_type;
                    dest_type dest;
                    composite_subscription lifetime;
                    select_type selector;

                    on_error_resume_next_observer(dest_type d, composite_subscription cs, select_type s)
                        : dest(std::move(d))
                        , lifetime(std::move(cs))
                        , selector(std::move(s))
                    {
                        dest.add(lifetime);
                    }
                    void on_next(value_type v) const {
                        dest.on_next(std::move(v));
                    }
                    void on_error(std::exception_ptr e) const {
                        auto selected = on_exception(
                            [&](){
                            return this->selector(std::move(e)); },
                                dest);
                            if (selected.empty()) {
                                return;
                            }
                            selected->subscribe(dest);
                    }
```

```
                                                void on_completed() const {
                                                        dest.on_completed();
                                                }

                                                static subscriber<T, observer_type> make(dest_type d, select_type s) {
                                                        auto cs = composite_subscription();
                                                        return make_subscriber<T>(cs, observer_type(this_type(std::move(d), cs,
std::move(s))));
                                                }
                                        };

                                        template<class Subscriber>
                                        auto operator()(Subscriber dest) const
                                                -> decltype(on_error_resume_next_observer<Subscriber>::make(std::move(dest),
selector)) {
                                                return     on_error_resume_next_observer<Subscriber>::make(std::move(dest), selector);
                                        }
                                };

                                template<class Selector>
                                class on_error_resume_next_factory
                                {
                                        typedef rxu::decay_t<Selector> select_type;
                                        select_type selector;
                                public:
                                        on_error_resume_next_factory(select_type s) : selector(std::move(s)) {}
                                        template<class Observable>
                                        auto operator()(Observable&& source)
                                                -> decltype(source.template
lift<rxu::value_type_t<on_error_resume_next<rxu::value_type_t<rxu::decay_t<Observable>>,
select_type>>>(on_error_resume_next<rxu::value_type_t<rxu::decay_t<Observable>>, select_type>(selector))) {
                                                return     source.template
lift<rxu::value_type_t<on_error_resume_next<rxu::value_type_t<rxu::decay_t<Observable>>,
select_type>>>(on_error_resume_next<rxu::value_type_t<rxu::decay_t<Observable>>, select_type>(selector));
                                        }
                                };

                        }

                        template<class Selector>
                        auto on_error_resume_next(Selector&& p)
                                ->      detail::on_error_resume_next_factory<Selector> {
                                return  detail::on_error_resume_next_factory<Selector>(std::forward<Selector>(p));
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_PAIRWISE_HPP)
#define RXCPP_OPERATORS_RX_PAIRWISE_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T>
                        struct pairwise
                        {
                                typedef rxu::decay_t<T> source_value_type;
                                typedef std::tuple<source_value_type, source_value_type> value_type;

                                template<class Subscriber>
                                struct pairwise_observer
                                {
                                        typedef pairwise_observer<Subscriber> this_type;
                                        typedef std::tuple<source_value_type, source_value_type> value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<T, this_type> observer_type;
                                        dest_type dest;
                                        mutable rxu::detail::maybe<source_value_type> remembered;

                                        pairwise_observer(dest_type d)
                                                : dest(std::move(d))
                                        {
```

```
                                        }
                                        void on_next(source_value_type v) const {
                                                if (remembered.empty()) {
                                                        remembered.reset(v);
                                                        return;
                                                }

                                                dest.on_next(std::make_tuple(remembered.get(), v));
                                                remembered.reset(v);
                                        }
                                        void on_error(std::exception_ptr e) const {
                                                dest.on_error(e);
                                        }
                                        void on_completed() const {
                                                dest.on_completed();
                                        }

                                        static subscriber<T, observer_type> make(dest_type d) {
                                                auto cs = d.get_subscription();
                                                return make_subscriber<T>(std::move(cs),
observer_type(this_type(std::move(d))));
                                        }
                                };

                                template<class Subscriber>
                                auto operator()(Subscriber dest) const
                                        -> decltype(pairwise_observer<Subscriber>::make(std::move(dest))) {
                                        return     pairwise_observer<Subscriber>::make(std::move(dest));
                                }
                        };

                        class pairwise_factory
                        {
                        public:
                                template<class Observable>
                                auto operator()(Observable&& source)
                                        -> decltype(source.template
lift<rxu::value_type_t<pairwise<rxu::value_type_t<rxu::decay_t<Observable>>>>>(pairwise<rxu::value_type_t<rxu::decay_t<Observable>>>())
) {
                                        return     source.template
lift<rxu::value_type_t<pairwise<rxu::value_type_t<rxu::decay_t<Observable>>>>>(pairwise<rxu::value_type_t<rxu::decay_t<Observable>>>())
;
                                }
                        };

                }

                inline auto pairwise()
                        ->      detail::pairwise_factory {
                        return  detail::pairwise_factory();
                }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_PUBLISH_HPP)
#define RXCPP_OPERATORS_RX_PUBLISH_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<template<class T> class Subject>
                        class publish_factory
                        {
                        public:
                                publish_factory() {}
                                template<class Observable>
                                auto operator()(Observable&& source)
                                        ->      connectable_observable<rxu::value_type_t<rxu::decay_t<Observable>>,
multicast<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, Subject<rxu::value_type_t<rxu::decay_t<Observable>>>>> {
                                        return  connectable_observable<rxu::value_type_t<rxu::decay_t<Observable>>,
multicast<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, Subject<rxu::value_type_t<rxu::decay_t<Observable>>>>(
```

```
                                                                multicast<rxu::value_type_t<rxu::decay_t<Observable>>, Observable,
Subject<rxu::value_type_t<rxu::decay_t<Observable>>>>(
                                                                std::forward<Observable>(source),
Subject<rxu::value_type_t<rxu::decay_t<Observable>>>()));
                                        }
                                };

                        }

                        inline auto publish()
                                ->      detail::publish_factory<rxsub::subject> {
                                return  detail::publish_factory<rxsub::subject>();
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_REF_COUNT_HPP)
#define RXCPP_OPERATORS_RX_REF_COUNT_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class ConnectableObservable>
                        struct ref_count : public operator_base<T>
                        {
                                typedef rxu::decay_t<ConnectableObservable> source_type;

                                struct ref_count_state : public std::enable_shared_from_this<ref_count_state>
                                {
                                        explicit ref_count_state(source_type o)
                                        : source(std::move(o))
                                        , subscribers(0)
                                        {
                                        }

                                        source_type source;
                                        std::mutex lock;
                                        long subscribers;
                                        composite_subscription connection;
                                };
                                std::shared_ptr<ref_count_state> state;

                                explicit ref_count(source_type o)
                                        : state(std::make_shared<ref_count_state>(std::move(o)))
                                {
                                }

                                template<class Subscriber>
                                void on_subscribe(Subscriber&& o) const {
                                        std::unique_lock<std::mutex> guard(state->lock);
                                        auto needConnect = ++state->subscribers == 1;
                                        auto keepAlive = state;
                                        guard.unlock();
                                        o.add(
                                                [keepAlive](){
                                                std::unique_lock<std::mutex> guard_unsubscribe(keepAlive->lock);
                                                if (--keepAlive->subscribers == 0) {
                                                        keepAlive->connection.unsubscribe();
                                                        keepAlive->connection = composite_subscription();
                                                }
                                        });
                                        keepAlive->source.subscribe(std::forward<Subscriber>(o));
                                        if (needConnect) {
                                                keepAlive->source.connect(keepAlive->connection);
                                        }
                                }
                        };

                        class ref_count_factory
                        {
                        public:
                                ref_count_factory() {}
```

```
                                        template<class... TN>
                                        auto operator()(connectable_observable<TN...>&& source)
                                                -> observable<rxu::value_type_t<connectable_observable<TN...>>,
ref_count<rxu::value_type_t<connectable_observable<TN...>>, connectable_observable<TN...>>> {
                                                return observable<rxu::value_type_t<connectable_observable<TN...>>,
ref_count<rxu::value_type_t<connectable_observable<TN...>>, connectable_observable<TN...>>>(
                                                        ref_count<rxu::value_type_t<connectable_observable<TN...>>,
connectable_observable<TN...>>(std::move(source)));
                                        }
                                        template<class... TN>
                                        auto operator()(const connectable_observable<TN...>& source)
                                                -> observable<rxu::value_type_t<connectable_observable<TN...>>,
ref_count<rxu::value_type_t<connectable_observable<TN...>>, connectable_observable<TN...>>> {
                                                return observable<rxu::value_type_t<connectable_observable<TN...>>,
ref_count<rxu::value_type_t<connectable_observable<TN...>>, connectable_observable<TN...>>>(
                                                        ref_count<rxu::value_type_t<connectable_observable<TN...>>,
connectable_observable<TN...>>(source));
                                        }
                                };

                        }

                        inline auto ref_count()
                                -> detail::ref_count_factory {
                                return detail::ref_count_factory();
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_REPEAT_HPP)
#define RXCPP_OPERATORS_RX_REPEAT_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Observable, class Count>
                        struct repeat : public operator_base<T>
                        {
                                typedef rxu::decay_t<Observable> source_type;
                                typedef rxu::decay_t<Count> count_type;
                                struct values
                                {
                                        values(source_type s, count_type t)
                                        : source(std::move(s))
                                        , remaining(std::move(t))
                                        , repeat_infinitely(t == 0)
                                        {
                                        }
                                        source_type source;
                                        count_type remaining;
                                        bool repeat_infinitely;
                                };
                                values initial;

                                repeat(source_type s, count_type t)
                                        : initial(std::move(s), std::move(t))
                                {
                                }

                                template<class Subscriber>
                                void on_subscribe(const Subscriber& s) const {

                                        typedef Subscriber output_type;
                                        struct state_type
                                                : public std::enable_shared_from_this<state_type>
                                                , public values
                                        {
                                                state_type(const values& i, const output_type& oarg)
                                                : values(i)
                                                , source_lifetime(composite_subscription::empty())
                                                , out(oarg)
                                                {
```

```
                                                }
                                                composite_subscription source_lifetime;
                                                output_type out;
                                                composite_subscription::weak_subscription lifetime_token;

                                                void do_subscribe() {
                                                        auto state = this->shared_from_this();

                                                        state->out.remove(state->lifetime_token);
                                                        state->source_lifetime.unsubscribe();

                                                        state->source_lifetime = composite_subscription();
                                                        state->lifetime_token = state->out.add(state->source_lifetime);

                                                        state->source.subscribe(
                                                                state->out,
                                                                state->source_lifetime,
                                                                // on_next
                                                                [state](T t) {
                                                                state->out.on_next(t);
                                                        },
                                                                // on_error
                                                                [state](std::exception_ptr e) {
                                                                state->out.on_error(e);
                                                        },
                                                                // on_completed
                                                                [state]() {
                                                                if (state->repeat_infinitely || (--state->remaining > 0)) {
                                                                        state->do_subscribe();
                                                                }
                                                                else {
                                                                        state->out.on_completed();
                                                                }
                                                        }
                                                        );
                                                }
                                        };

                                        // take a copy of the values for each subscription
                                        auto state = std::make_shared<state_type>(initial, s);

                                        // start the first iteration
                                        state->do_subscribe();
                                }
                        };

                        template<class T>
                        class repeat_factory
                        {
                                typedef rxu::decay_t<T> count_type;
                                count_type count;
                        public:
                                repeat_factory(count_type t) : count(std::move(t)) {}

                                template<class Observable>
                                auto operator()(Observable&& source)
                                        ->      observable<rxu::value_type_t<rxu::decay_t<Observable>>,
repeat<rxu::value_type_t<rxu::decay_t<Observable>>, rxu::decay_t<Observable>, count_type>> {
                                        return  observable<rxu::value_type_t<rxu::decay_t<Observable>>,
repeat<rxu::value_type_t<rxu::decay_t<Observable>>, rxu::decay_t<Observable>, count_type>>(
                                                repeat<rxu::value_type_t<rxu::decay_t<Observable>>,
rxu::decay_t<Observable>, count_type>(std::forward<Observable>(source), count));
                                }
                        };

                }

                template<class T>
                auto repeat(T&& t)
                        ->      detail::repeat_factory<T> {
                        return  detail::repeat_factory<T>(std::forward<T>(t));
                }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_REPLAY_HPP)
#define RXCPP_OPERATORS_RX_REPLAY_HPP
```

```
//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace operators {

		namespace detail {

			template<template<class T, class Coordination> class Subject, class Coordination>
			class replay_factory
			{
				typedef rxu::decay_t<Coordination> coordination_type;

				coordination_type coordination;

			public:
				replay_factory(coordination_type cn)
					: coordination(std::move(cn))
				{
				}

				template<class Observable>
				auto operator()(Observable&& source)
					->      connectable_observable<rxu::value_type_t<rxu::decay_t<Observable>>,
multicast<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, Subject<rxu::value_type_t<rxu::decay_t<Observable>>, Coordination>>>
{
					return  connectable_observable<rxu::value_type_t<rxu::decay_t<Observable>>,
multicast<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, Subject<rxu::value_type_t<rxu::decay_t<Observable>>,
Coordination>>>(
						multicast<rxu::value_type_t<rxu::decay_t<Observable>>, Observable,
Subject<rxu::value_type_t<rxu::decay_t<Observable>>, Coordination>>(
						std::forward<Observable>(source),
Subject<rxu::value_type_t<rxu::decay_t<Observable>>, Coordination>(coordination)));
				}
			};

		}

		template<class Coordination>
		inline auto replay(Coordination&& cn)
			->      detail::replay_factory<rxsub::replay, Coordination> {
			return  detail::replay_factory<rxsub::replay, Coordination>(std::forward<Coordination>(cn));
		}

	}

}

#endif


#if !defined(RXCPP_OPERATORS_RX_RETRY_HPP)
#define RXCPP_OPERATORS_RX_RETRY_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace operators {

		namespace detail {

			template<class T, class Observable, class Count>
			struct retry : public operator_base<T> {
				typedef rxu::decay_t<Observable> source_type;
				typedef rxu::decay_t<Count> count_type;
				struct values {
					values(source_type s, count_type t)
					: source(std::move(s))
					, remaining(std::move(t))
					, retry_infinitely(t == 0) {
					}
					source_type source;
					count_type remaining;
					bool retry_infinitely;
				};
				values initial;

				retry(source_type s, count_type t)
					: initial(std::move(s), std::move(t)) {
				}
```

```
                template<class Subscriber>
                void on_subscribe(const Subscriber& s) const {

                    typedef Subscriber output_type;
                    struct state_type
                        : public std::enable_shared_from_this<state_type>
                        , public values {
                        state_type(const values& i, const output_type& oarg)
                        : values(i)
                        , source_lifetime(composite_subscription::empty())
                        , out(oarg) {
                        }
                        composite_subscription source_lifetime;
                        output_type out;

                        void do_subscribe() {
                            auto state = this->shared_from_this();

                            state->source_lifetime = composite_subscription();
                            state->out.add(state->source_lifetime);

                            state->source.subscribe(
                                state->out,
                                state->source_lifetime,
                                // on_next
                                [state](T t) {
                                state->out.on_next(t);
                            },
                                // on_error
                                [state](std::exception_ptr e) {
                                if (state->retry_infinitely || (--state->remaining >= 0)) {
                                    state->do_subscribe();
                                }
                                else {
                                    state->out.on_error(e);
                                }
                            },
                                // on_completed
                                [state]() {

                                // JEP: never appeears to be called?

                                state->out.on_completed();
                            }
                            );
                        }
                    };

                    // take a copy of the values for each subscription
                    auto state = std::make_shared<state_type>(initial, s);

                    // start the first iteration
                    state->do_subscribe();
                }
            };

            template<class T>
            class retry_factory {
                typedef rxu::decay_t<T> count_type;
                count_type count;
            public:
                retry_factory(count_type t) : count(std::move(t)) {}

                template<class Observable>
                auto operator()(Observable&& source)
                    ->  observable<rxu::value_type_t<rxu::decay_t<Observable>>,
retry<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, count_type>> {
                    return  observable<rxu::value_type_t<rxu::decay_t<Observable>>,
retry<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, count_type>>(
                        retry<rxu::value_type_t<rxu::decay_t<Observable>>, Observable,
count_type>(std::forward<Observable>(source), count));
                }
            };

        }

        template<class T>
        auto retry(T&& t)
            ->  detail::retry_factory<T> {
            return  detail::retry_factory<T>(std::forward<T>(t));
        }
```

```
		}

}

#endif


#if !defined(RXCPP_OPERATORS_RX_SAMPLE_WITH_TIME_HPP)
#define RXCPP_OPERATORS_RX_SAMPLE_WITH_TIME_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace operators {

		namespace detail {

			template<class T, class Duration, class Coordination>
			struct sample_with_time
			{
				static_assert(std::is_convertible<Duration, rxsc::scheduler::clock_type::duration>::value, "Duration parameter must convert to rxsc::scheduler::clock_type::duration");
				static_assert(is_coordination<Coordination>::value, "Coordination parameter must satisfy the requirements for a Coordination");

				typedef rxu::decay_t<T> source_value_type;
				typedef rxu::decay_t<Coordination> coordination_type;
				typedef typename coordination_type::coordinator_type coordinator_type;
				typedef rxu::decay_t<Duration> duration_type;

				struct sample_with_time_value
				{
					sample_with_time_value(duration_type p, coordination_type c)
					: period(p)
					, coordination(c)
					{
					}
					duration_type period;
					coordination_type coordination;
				};
				sample_with_time_value initial;

				sample_with_time(duration_type period, coordination_type coordination)
					: initial(period, coordination)
				{
				}

				template<class Subscriber>
				struct sample_with_time_observer
				{
					typedef sample_with_time_observer<Subscriber> this_type;
					typedef T value_type;
					typedef rxu::decay_t<Subscriber> dest_type;
					typedef observer<value_type, this_type> observer_type;

					struct sample_with_time_subscriber_value : public sample_with_time_value
					{
						sample_with_time_subscriber_value(composite_subscription cs, dest_type d, sample_with_time_value v, coordinator_type c)
						: sample_with_time_value(v)
						, cs(std::move(cs))
						, dest(std::move(d))
						, coordinator(std::move(c))
						, worker(coordinator.get_worker())
						{
						}
						composite_subscription cs;
						dest_type dest;
						coordinator_type coordinator;
						rxsc::worker worker;
						mutable rxu::maybe<value_type> value;
					};
					std::shared_ptr<sample_with_time_subscriber_value> state;

					sample_with_time_observer(composite_subscription cs, dest_type d, sample_with_time_value v, coordinator_type c)
						:
state(std::make_shared<sample_with_time_subscriber_value>(sample_with_time_subscriber_value(std::move(cs), std::move(d), v, std::move(c))))
					{
```

```
                auto localState = state;

                auto disposer = [=](const rxsc::schedulable&){
                        localState->cs.unsubscribe();
                        localState->dest.unsubscribe();
                        localState->worker.unsubscribe();
                };
                auto selectedDisposer = on_exception(
                        [&](){ return localState->coordinator.act(disposer); },
                        localState->dest);
                if (selectedDisposer.empty()) {
                        return;
                }

                localState->dest.add([=](){
                        localState->worker.schedule(selectedDisposer.get());
                });
                localState->cs.add([=](){
                        localState->worker.schedule(selectedDisposer.get());
                });

                auto produce_sample = [localState](const rxsc::schedulable&) {
                        if (!localState->value.empty()) {
                                localState->dest.on_next(*localState->value);
                                localState->value.reset();
                        }
                };
                auto selectedProduce = on_exception(
                        [&](){ return localState->coordinator.act(produce_sample); },
                        localState->dest);
                if (selectedProduce.empty()) {
                        return;
                }

                state->worker.schedule_periodically(
                        localState->worker.now(),
                        localState->period,
                        [localState, selectedProduce](const rxsc::schedulable&) {
                        localState->worker.schedule(selectedProduce.get());
                });
        }

        void on_next(T v) const {
                auto localState = state;
                auto work = [v, localState](const rxsc::schedulable&) {
                        localState->value.reset(v);
                };
                auto selectedWork = on_exception(
                        [&](){ return localState->coordinator.act(work); },
                        localState->dest);
                if (selectedWork.empty()) {
                        return;
                }
                localState->worker.schedule(selectedWork.get());
        }

        void on_error(std::exception_ptr e) const {
                auto localState = state;
                auto work = [e, localState](const rxsc::schedulable&) {
                        localState->dest.on_error(e);
                };
                auto selectedWork = on_exception(
                        [&](){ return localState->coordinator.act(work); },
                        localState->dest);
                if (selectedWork.empty()) {
                        return;
                }
                localState->worker.schedule(selectedWork.get());
        }

        void on_completed() const {
                auto localState = state;
                auto work = [localState](const rxsc::schedulable&) {
                        localState->dest.on_completed();
                };
                auto selectedWork = on_exception(
                        [&](){ return localState->coordinator.act(work); },
                        localState->dest);
                if (selectedWork.empty()) {
                        return;
                }
                localState->worker.schedule(selectedWork.get());
```

```
                                        }

                                        static subscriber<T, observer<T, this_type>> make(dest_type d, sample_with_time_value
v) {
                                                auto cs = composite_subscription();
                                                auto coordinator = v.coordination.create_coordinator();

                                                return make_subscriber<T>(cs, this_type(cs, std::move(d), std::move(v),
std::move(coordinator)));
                                        }
                                };

                                template<class Subscriber>
                                auto operator()(Subscriber dest) const
                                        -> decltype(sample_with_time_observer<Subscriber>::make(std::move(dest), initial)) {
                                        return      sample_with_time_observer<Subscriber>::make(std::move(dest), initial);
                                }
                        };

                        template<class Duration, class Coordination>
                        class sample_with_time_factory
                        {
                                typedef rxu::decay_t<Duration> duration_type;
                                typedef rxu::decay_t<Coordination> coordination_type;

                                duration_type period;
                                coordination_type coordination;
                        public:
                                sample_with_time_factory(duration_type p, coordination_type c) : period(p), coordination(c) {}
                                template<class Observable>
                                auto operator()(Observable&& source)
                                        -> decltype(source.template
lift<rxu::value_type_t<rxu::decay_t<Observable>>>(sample_with_time<rxu::value_type_t<rxu::decay_t<Observable>>, Duration,
Coordination>(period, coordination))) {
                                        return      source.template
lift<rxu::value_type_t<rxu::decay_t<Observable>>>(sample_with_time<rxu::value_type_t<rxu::decay_t<Observable>>, Duration,
Coordination>(period, coordination));
                                }
                        };

                }

                template<class Duration, class Coordination>
                inline auto sample_with_time(Duration period, Coordination coordination)
                        ->      detail::sample_with_time_factory<Duration, Coordination> {
                        return  detail::sample_with_time_factory<Duration, Coordination>(period, coordination);
                }

                template<class Duration>
                inline auto sample_with_time(Duration period)
                        ->      detail::sample_with_time_factory<Duration, identity_one_worker> {
                        return  detail::sample_with_time_factory<Duration, identity_one_worker>(period, identity_current_thread());
                }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_SCAN_HPP)
#define RXCPP_OPERATORS_RX_SCAN_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Observable, class Accumulator, class Seed>
                        struct scan : public operator_base<rxu::decay_t<Seed>>
                        {
                                typedef rxu::decay_t<Observable> source_type;
                                typedef rxu::decay_t<Accumulator> accumulator_type;
                                typedef rxu::decay_t<Seed> seed_type;

                                struct scan_initial_type
                                {
                                        scan_initial_type(source_type o, accumulator_type a, seed_type s)
```

```
                                        : source(std::move(o))
                                        , accumulator(std::move(a))
                                        , seed(s)
                                        {
                                        }
                                        source_type source;
                                        accumulator_type accumulator;
                                        seed_type seed;
                                };
                                scan_initial_type initial;

                                template<class CT, class CS, class CP>
                                static auto check(int) -> decltype((*(CP*)nullptr)(*(CS*)nullptr, *(CT*)nullptr));
                                template<class CT, class CS, class CP>
                                static void check(...);

                                scan(source_type o, accumulator_type a, seed_type s)
                                        : initial(std::move(o), a, s)
                                {
                                        static_assert(std::is_convertible<decltype(check<T, seed_type, accumulator_type>(0)),
seed_type>::value, "scan Accumulator must be a function with the signature Seed(Seed, T)");
                                }
                                template<class Subscriber>
                                void on_subscribe(Subscriber o) const {
                                        struct scan_state_type
                                                : public scan_initial_type
                                                , public std::enable_shared_from_this<scan_state_type>
                                        {
                                                scan_state_type(scan_initial_type i, Subscriber scrbr)
                                                : scan_initial_type(i)
                                                , result(scan_initial_type::seed)
                                                , out(std::move(scrbr))
                                                {
                                                }
                                                seed_type result;
                                                Subscriber out;
                                        };
                                        auto state = std::make_shared<scan_state_type>(initial, std::move(o));
                                        state->source.subscribe(
                                                state->out,
                                                // on_next
                                                [state](T t) {
                                                state->result = state->accumulator(state->result, t);
                                                state->out.on_next(state->result);
                                        },
                                                // on_error
                                                [state](std::exception_ptr e) {
                                                state->out.on_error(e);
                                        },
                                                // on_completed
                                                [state]() {
                                                state->out.on_completed();
                                        }
                                        );
                                }
                        };

                        template<class Accumulator, class Seed>
                        class scan_factory
                        {
                                typedef rxu::decay_t<Accumulator> accumulator_type;
                                typedef rxu::decay_t<Seed> seed_type;

                                accumulator_type accumulator;
                                seed_type seed;
                        public:
                                scan_factory(accumulator_type a, Seed s)
                                        : accumulator(std::move(a))
                                        , seed(s)
                                {
                                }
                                template<class Observable>
                                auto operator()(Observable&& source)
                                        ->      observable<rxu::decay_t<Seed>,
scan<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, Accumulator, Seed>> {
                                        return  observable<rxu::decay_t<Seed>,
scan<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, Accumulator, Seed>>(
                                                scan<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, Accumulator,
Seed>(std::forward<Observable>(source), accumulator, seed));
                                }
                        };
```

```
                }

                template<class Seed, class Accumulator>
                auto scan(Seed s, Accumulator&& a)
                        ->      detail::scan_factory<Accumulator, Seed> {
                        return  detail::scan_factory<Accumulator, Seed>(std::forward<Accumulator>(a), s);
                }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_SEQUENCE_EQUAL_HPP)
#define RXCPP_OPERATORS_RX_SEQUENCE_EQUAL_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Observable, class OtherObservable, class BinaryPredicate, class Coordination>
                        struct sequence_equal : public operator_base<bool>
                        {
                                typedef rxu::decay_t<Observable> source_type;
                                typedef rxu::decay_t<T> source_value_type;
                                typedef rxu::decay_t<OtherObservable> other_source_type;
                                typedef typename other_source_type::value_type other_source_value_type;
                                typedef rxu::decay_t<BinaryPredicate> predicate_type;
                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef typename coordination_type::coordinator_type coordinator_type;

                                struct values {
                                        values(source_type s, other_source_type t, predicate_type pred, coordination_type sf)
                                        : source(std::move(s))
                                        , other(std::move(t))
                                        , pred(std::move(pred))
                                        , coordination(std::move(sf))
                                        {
                                        }

                                        source_type source;
                                        other_source_type other;
                                        predicate_type pred;
                                        coordination_type coordination;
                                };

                                values initial;

                                sequence_equal(source_type s, other_source_type t, predicate_type pred, coordination_type sf)
                                        : initial(std::move(s), std::move(t), std::move(pred), std::move(sf))
                                {
                                }

                                template<class Subscriber>
                                void on_subscribe(Subscriber s) const {

                                        typedef Subscriber output_type;

                                        struct state_type
                                                : public std::enable_shared_from_this<state_type>
                                                , public values
                                        {
                                                state_type(const values& vals, coordinator_type coor, const output_type& o)
                                                : values(vals)
                                                , coordinator(std::move(coor))
                                                , out(o)
                                                , source_completed(false)
                                                , other_completed(false)
                                                {
                                                        out.add(other_lifetime);
                                                        out.add(source_lifetime);
                                                }

                                                composite_subscription other_lifetime;
                                                composite_subscription source_lifetime;
                                                coordinator_type coordinator;
```

```
                output_type out;

                mutable std::list<source_value_type> source_values;
                mutable std::list<other_source_value_type> other_values;
                mutable bool source_completed;
                mutable bool other_completed;
        };

        auto coordinator = initial.coordination.create_coordinator();
        auto state = std::make_shared<state_type>(initial, std::move(coordinator), std::move(s));

        auto other = on_exception(
                [&](){ return state->coordinator.in(state->other); },
                state->out);
        if (other.empty()) {
                return;
        }

        auto source = on_exception(
                [&](){ return state->coordinator.in(state->source); },
                state->out);
        if (source.empty()) {
                return;
        }

        auto check_equal = [state]() {
                if (!state->source_values.empty() && !state->other_values.empty()) {
                        auto x = std::move(state->source_values.front());
                        state->source_values.pop_front();

                        auto y = std::move(state->other_values.front());
                        state->other_values.pop_front();

                        if (!state->pred(x, y)) {
                                state->out.on_next(false);
                                state->out.on_completed();
                        }
                }
                else {
                        if ((!state->source_values.empty() && state->other_completed) ||
                                (!state->other_values.empty() && state-
>source_completed)) {

                                state->out.on_next(false);
                                state->out.on_completed();
                        }
                }
        };

        auto check_complete = [state]() {
                if (state->source_completed && state->other_completed) {
                        state->out.on_next(state->source_values.empty() && state-
>other_values.empty());

                        state->out.on_completed();
                }
        };

        auto sinkOther = make_subscriber<other_source_value_type>(
                state->out,
                state->other_lifetime,
                // on_next
                [state, check_equal](other_source_value_type t) {
                auto& values = state->other_values;
                values.push_back(t);
                check_equal();
        },
                // on_error
                [state](std::exception_ptr e) {
                state->out.on_error(e);
        },
                // on_completed
                [state, check_complete]() {
                auto& completed = state->other_completed;
                completed = true;
                check_complete();
        }
        );

        auto selectedSinkOther = on_exception(
                [&](){ return state->coordinator.out(sinkOther); },
                state->out);
        if (selectedSinkOther.empty()) {
                return;
```

```
                                        }
                                        other->subscribe(std::move(selectedSinkOther.get()));

                                        source.get().subscribe(
                                                state->source_lifetime,
                                                // on_next
                                                [state, check_equal](source_value_type t) {
                                                auto& values = state->source_values;
                                                values.push_back(t);
                                                check_equal();
                                        },
                                                // on_error
                                                [state](std::exception_ptr e) {
                                                state->out.on_error(e);
                                        },
                                                // on_completed
                                                [state, check_complete]() {
                                                auto& completed = state->source_completed;
                                                completed = true;
                                                check_complete();
                                        }
                                        );
                                }
                        };

                        template<class OtherObservable, class BinaryPredicate, class Coordination>
                        class sequence_equal_factory
                        {
                                typedef rxu::decay_t<OtherObservable> other_source_type;
                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef rxu::decay_t<BinaryPredicate> predicate_type;

                                other_source_type other_source;
                                coordination_type coordination;
                                predicate_type pred;

                        public:
                                sequence_equal_factory(other_source_type t, predicate_type p, coordination_type sf)
                                        : other_source(std::move(t))
                                        , coordination(std::move(sf))
                                        , pred(std::move(p))
                                {
                                }

                                template<class Observable>
                                auto operator()(Observable&& source)
                                        ->      observable<bool, sequence_equal<rxu::value_type_t<rxu::decay_t<Observable>>,
Observable, other_source_type, BinaryPredicate, Coordination>> {
                                        return  observable<bool, sequence_equal<rxu::value_type_t<rxu::decay_t<Observable>>,
Observable, other_source_type, BinaryPredicate, Coordination>>(
                                                sequence_equal<rxu::value_type_t<rxu::decay_t<Observable>>, Observable,
other_source_type, BinaryPredicate, Coordination>(std::forward<Observable>(source), other_source, pred, coordination));
                                }
                        };

                }

                template<class OtherObservable>
                inline auto sequence_equal(OtherObservable&& t)
                        ->      detail::sequence_equal_factory<OtherObservable, rxu::equal_to<>, identity_one_worker> {
                        return  detail::sequence_equal_factory<OtherObservable, rxu::equal_to<>,
identity_one_worker>(std::forward<OtherObservable>(t), rxu::equal_to<>(), identity_current_thread());
                }

                template<class OtherObservable, class BinaryPredicate, class Check = typename
std::enable_if<!is_coordination<BinaryPredicate>::value>::type>
                inline auto sequence_equal(OtherObservable&& t, BinaryPredicate&& pred)
                        ->      detail::sequence_equal_factory<OtherObservable, BinaryPredicate, identity_one_worker> {
                        return  detail::sequence_equal_factory<OtherObservable, BinaryPredicate,
identity_one_worker>(std::forward<OtherObservable>(t), std::forward<BinaryPredicate>(pred), identity_current_thread());
                }

                template<class OtherObservable, class Coordination, class Check = typename
std::enable_if<is_coordination<Coordination>::value>::type>
                inline auto sequence_equal(OtherObservable&& t, Coordination&& cn)
                        ->      detail::sequence_equal_factory<OtherObservable, rxu::equal_to<>, Coordination> {
                        return  detail::sequence_equal_factory<OtherObservable, rxu::equal_to<>,
Coordination>(std::forward<OtherObservable>(t), rxu::equal_to<>(), std::forward<Coordination>(cn));
                }

                template<class OtherObservable, class BinaryPredicate, class Coordination>
                inline auto sequence_equal(OtherObservable&& t, BinaryPredicate&& pred, Coordination&& cn)
```

```
                                        ->      detail::sequence_equal_factory<OtherObservable, BinaryPredicate, Coordination> {
                                        return  detail::sequence_equal_factory<OtherObservable, BinaryPredicate,
Coordination>(std::forward<OtherObservable>(t), std::forward<BinaryPredicate>(pred), std::forward<Coordination>(cn));
                                }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_SKIP_HPP)
#define RXCPP_OPERATORS_RX_SKIP_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Observable, class Count>
                        struct skip : public operator_base<T>
                        {
                                typedef rxu::decay_t<Observable> source_type;
                                typedef rxu::decay_t<Count> count_type;
                                struct values
                                {
                                        values(source_type s, count_type t)
                                        : source(std::move(s))
                                        , count(std::move(t))
                                        {
                                        }
                                        source_type source;
                                        count_type count;
                                };
                                values initial;

                                skip(source_type s, count_type t)
                                        : initial(std::move(s), std::move(t))
                                {
                                }

                                struct mode
                                {
                                        enum type {
                                                skipping,  // ignore messages
                                                triggered, // capture messages
                                                errored,   // error occured
                                                stopped    // observable completed
                                        };
                                };

                                template<class Subscriber>
                                void on_subscribe(const Subscriber& s) const {

                                        typedef Subscriber output_type;
                                        struct state_type
                                                : public std::enable_shared_from_this<state_type>
                                                , public values
                                        {
                                                state_type(const values& i, const output_type& oarg)
                                                : values(i)
                                                , mode_value(i.count > 0 ? mode::skipping : mode::triggered)
                                                , out(oarg)
                                                {
                                                }
                                                typename mode::type mode_value;
                                                output_type out;
                                        };
                                        // take a copy of the values for each subscription
                                        auto state = std::make_shared<state_type>(initial, s);

                                        composite_subscription source_lifetime;

                                        s.add(source_lifetime);

                                        state->source.subscribe(
                                                // split subscription lifetime
                                                source_lifetime,
```

```
                                                // on_next
                                                [state](T t) {
                                                if (state->mode_value == mode::skipping) {
                                                        if (--state->count == 0) {
                                                                state->mode_value = mode::triggered;
                                                        }
                                                }
                                                else {
                                                        state->out.on_next(t);
                                                }
                                        },
                                                // on_error
                                                [state](std::exception_ptr e) {
                                                state->mode_value = mode::errored;
                                                state->out.on_error(e);
                                        },
                                                // on_completed
                                                [state]() {
                                                state->mode_value = mode::stopped;
                                                state->out.on_completed();
                                        }
                                        );
                                }
                        };

                        template<class T>
                        class skip_factory
                        {
                                typedef rxu::decay_t<T> count_type;
                                count_type count;
                        public:
                                skip_factory(count_type t) : count(std::move(t)) {}
                                template<class Observable>
                                auto operator()(Observable&& source)
                                        ->      observable<rxu::value_type_t<rxu::decay_t<Observable>>,
skip<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, count_type>> {
                                        return  observable<rxu::value_type_t<rxu::decay_t<Observable>>,
skip<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, count_type>>(
                                                skip<rxu::value_type_t<rxu::decay_t<Observable>>, Observable,
count_type>(std::forward<Observable>(source), count));
                                }
                        };

                }

                template<class T>
                auto skip(T&& t)
                        ->      detail::skip_factory<T> {
                        return  detail::skip_factory<T>(std::forward<T>(t));
                }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_SKIP_LAST_HPP)
#define RXCPP_OPERATORS_RX_SKIP_LAST_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Observable, class Count>
                        struct skip_last : public operator_base<T>
                        {
                                typedef rxu::decay_t<Observable> source_type;
                                typedef rxu::decay_t<Count> count_type;

                                typedef std::queue<T> queue_type;
                                typedef typename queue_type::size_type queue_size_type;

                                struct values
                                {
                                        values(source_type s, count_type t)
                                        : source(std::move(s))
```

```
                                        , count(static_cast<queue_size_type>(t))
                                        {
                                        }
                                        source_type source;
                                        queue_size_type count;
                                };
                                values initial;

                                skip_last(source_type s, count_type t)
                                        : initial(std::move(s), std::move(t))
                                {
                                }

                                template<class Subscriber>
                                void on_subscribe(const Subscriber& s) const {

                                        typedef Subscriber output_type;
                                        struct state_type
                                                : public std::enable_shared_from_this<state_type>
                                                , public values
                                        {
                                                state_type(const values& i, const output_type& oarg)
                                                : values(i)
                                                , out(oarg)
                                                {
                                                }
                                                queue_type items;
                                                output_type out;
                                        };
                                        // take a copy of the values for each subscription
                                        auto state = std::make_shared<state_type>(initial, s);

                                        composite_subscription source_lifetime;

                                        s.add(source_lifetime);

                                        state->source.subscribe(
                                                // split subscription lifetime
                                                source_lifetime,
                                                // on_next
                                                [state](T t) {
                                                if (state->count > 0) {
                                                        if (state->items.size() == state->count) {
                                                                state->out.on_next(std::move(state->items.front()));
                                                                state->items.pop();
                                                        }
                                                        state->items.push(t);
                                                }
                                                else {
                                                        state->out.on_next(t);
                                                }
                                        },
                                                // on_error
                                                [state](std::exception_ptr e) {
                                                state->out.on_error(e);
                                        },
                                                // on_completed
                                                [state]() {
                                                state->out.on_completed();
                                        }
                                        );
                                }
                        };

                        template<class T>
                        class skip_last_factory
                        {
                                typedef rxu::decay_t<T> count_type;
                                count_type count;
                        public:
                                skip_last_factory(count_type t) : count(std::move(t)) {}
                                template<class Observable>
                                auto operator()(Observable&& source)
                                        ->      observable<rxu::value_type_t<rxu::decay_t<Observable>>,
skip_last<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, count_type>> {
                                        return  observable<rxu::value_type_t<rxu::decay_t<Observable>>,
skip_last<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, count_type>>(
                                                skip_last<rxu::value_type_t<rxu::decay_t<Observable>>, Observable,
count_type>(std::forward<Observable>(source), count));
                                }
                        };
```

```
                }

                template<class T>
                auto skip_last(T&& t)
                        ->      detail::skip_last_factory<T> {
                        return  detail::skip_last_factory<T>(std::forward<T>(t));
                }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_SKIP_UNTIL_HPP)
#define RXCPP_OPERATORS_RX_SKIP_UNTIL_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Observable, class TriggerObservable, class Coordination>
                        struct skip_until : public operator_base<T>
                        {
                                typedef rxu::decay_t<Observable> source_type;
                                typedef rxu::decay_t<TriggerObservable> trigger_source_type;
                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef typename coordination_type::coordinator_type coordinator_type;
                                struct values
                                {
                                        values(source_type s, trigger_source_type t, coordination_type sf)
                                        : source(std::move(s))
                                        , trigger(std::move(t))
                                        , coordination(std::move(sf))
                                        {
                                        }
                                        source_type source;
                                        trigger_source_type trigger;
                                        coordination_type coordination;
                                };
                                values initial;

                                skip_until(source_type s, trigger_source_type t, coordination_type sf)
                                        : initial(std::move(s), std::move(t), std::move(sf))
                                {
                                }

                                struct mode
                                {
                                        enum type {
                                                skipping,  // no messages from trigger
                                                clear,     // trigger completed
                                                triggered, // trigger sent on_next
                                                errored,   // error either on trigger or on observable
                                                stopped    // observable completed
                                        };
                                };

                                template<class Subscriber>
                                void on_subscribe(Subscriber s) const {

                                        typedef Subscriber output_type;
                                        struct state_type
                                                : public std::enable_shared_from_this<state_type>
                                                , public values
                                        {
                                                state_type(const values& i, coordinator_type coor, const output_type& oarg)
                                                : values(i)
                                                , mode_value(mode::skipping)
                                                , coordinator(std::move(coor))
                                                , out(oarg)
                                                {
                                                        out.add(trigger_lifetime);
                                                        out.add(source_lifetime);
                                                }
                                                typename mode::type mode_value;
                                                composite_subscription trigger_lifetime;
```

```
                composite_subscription source_lifetime;
                coordinator_type coordinator;
                output_type out;
};

auto coordinator = initial.coordination.create_coordinator();

// take a copy of the values for each subscription
auto state = std::make_shared<state_type>(initial, std::move(coordinator), std::move(s));

auto trigger = on_exception(
                [&](){return state->coordinator.in(state->trigger); },
                state->out);
if (trigger.empty()) {
                return;
}

auto source = on_exception(
                [&](){return state->coordinator.in(state->source); },
                state->out);
if (source.empty()) {
                return;
}

auto sinkTrigger = make_subscriber<typename trigger_source_type::value_type>(
                // share parts of subscription
                state->out,
                // new lifetime
                state->trigger_lifetime,
                // on_next
                [state](const typename trigger_source_type::value_type&) {
                if (state->mode_value != mode::skipping) {
                                return;
                }
                state->mode_value = mode::triggered;
                state->trigger_lifetime.unsubscribe();
},
                // on_error
                [state](std::exception_ptr e) {
                if (state->mode_value != mode::skipping) {
                                return;
                }
                state->mode_value = mode::errored;
                state->out.on_error(e);
},
                // on_completed
                [state]() {
                if (state->mode_value != mode::skipping) {
                                return;
                }
                state->mode_value = mode::clear;
                state->trigger_lifetime.unsubscribe();
}
);
auto selectedSinkTrigger = on_exception(
                [&](){return state->coordinator.out(sinkTrigger); },
                state->out);
if (selectedSinkTrigger.empty()) {
                return;
}
trigger->subscribe(std::move(selectedSinkTrigger.get()));

source.get().subscribe(
                // split subscription lifetime
                state->source_lifetime,
                // on_next
                [state](T t) {
                if (state->mode_value != mode::triggered) {
                                return;
                }
                state->out.on_next(t);
},
                // on_error
                [state](std::exception_ptr e) {
                if (state->mode_value > mode::triggered) {
                                return;
                }
                state->mode_value = mode::errored;
                state->out.on_error(e);
},
                // on_completed
                [state]() {
```

```
                                                if (state->mode_value != mode::triggered) {
                                                        return;
                                                }
                                                state->mode_value = mode::stopped;
                                                state->out.on_completed();
                                        }
                                        );
                                }
                        };

                        template<class TriggerObservable, class Coordination>
                        class skip_until_factory
                        {
                                typedef rxu::decay_t<TriggerObservable> trigger_source_type;
                                typedef rxu::decay_t<Coordination> coordination_type;

                                trigger_source_type trigger_source;
                                coordination_type coordination;
                        public:
                                skip_until_factory(trigger_source_type t, coordination_type sf)
                                        : trigger_source(std::move(t))
                                        , coordination(std::move(sf))
                                {
                                }
                                template<class Observable>
                                auto operator()(Observable&& source)
                                        ->      observable<rxu::value_type_t<rxu::decay_t<Observable>>,
skip_until<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, trigger_source_type, Coordination>> {
                                        return  observable<rxu::value_type_t<rxu::decay_t<Observable>>,
skip_until<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, trigger_source_type, Coordination>>(
                                                skip_until<rxu::value_type_t<rxu::decay_t<Observable>>, Observable,
trigger_source_type, Coordination>(std::forward<Observable>(source), trigger_source, coordination));
                                }
                        };

                }

                template<class TriggerObservable, class Coordination>
                auto skip_until(TriggerObservable&& t, Coordination&& sf)
                        ->      detail::skip_until_factory<TriggerObservable, Coordination> {
                        return  detail::skip_until_factory<TriggerObservable, Coordination>(std::forward<TriggerObservable>(t),
std::forward<Coordination>(sf));
                }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_START_WITH_HPP)
#define RXCPP_OPERATORS_RX_START_WITH_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class StartObservable>
                        class start_with_factory
                        {
                        public:
                                using start_type = rxu::decay_t<StartObservable>;

                                start_type start;

                                explicit start_with_factory(start_type s) : start(s) {}

                                template<class Observable>
                                auto operator()(Observable source)
                                        -> decltype(start.concat(source)) {
                                        return      start.concat(source);
                                }
                        };

                }

                template<class Value0, class... ValueN>
```

```
                auto start_with(Value0 v0, ValueN... vn)
                        ->      detail::start_with_factory<decltype(rxs::from(rxu::decay_t<Value0>(v0), rxu::decay_t<Value0>(vn)...))> {
                        return detail::start_with_factory<decltype(rxs::from(rxu::decay_t<Value0>(v0), rxu::decay_t<Value0>(vn)...))>(
                                rxs::from(rxu::decay_t<Value0>(v0), rxu::decay_t<Value0>(vn)...));
                }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_SUBSCRIBE_HPP)
#define RXCPP_OPERATORS_RX_SUBSCRIBE_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class Subscriber>
                        class subscribe_factory;

                        template<class T, class I>
                        class subscribe_factory<subscriber<T, I>>
                        {
                                subscriber<T, I> scrbr;
                        public:
                                subscribe_factory(subscriber<T, I> s)
                                        : scrbr(std::move(s))
                                {}
                                template<class Observable>
                                auto operator()(Observable&& source)
                                        -> decltype(std::forward<Observable>(source).subscribe(std::move(scrbr))) {
                                        return      std::forward<Observable>(source).subscribe(std::move(scrbr));
                                }
                        };

                }

                template<class T, class Arg0>
                auto subscribe(Arg0&& a0)
                        ->      detail::subscribe_factory<decltype  (make_subscriber<T>(std::forward<Arg0>(a0)))> {
                        return  detail::subscribe_factory<decltype  (make_subscriber<T>(std::forward<Arg0>(a0)))>
                                (make_subscriber<T>(std::forward<Arg0>(a0)));
                }
                template<class T, class Arg0, class Arg1>
                auto subscribe(Arg0&& a0, Arg1&& a1)
                        ->      detail::subscribe_factory<decltype  (make_subscriber<T>(std::forward<Arg0>(a0),
std::forward<Arg1>(a1)))> {
                        return  detail::subscribe_factory<decltype  (make_subscriber<T>(std::forward<Arg0>(a0),
std::forward<Arg1>(a1)))>
                                (make_subscriber<T>(std::forward<Arg0>(a0), std::forward<Arg1>(a1)));
                }
                template<class T, class Arg0, class Arg1, class Arg2>
                auto subscribe(Arg0&& a0, Arg1&& a1, Arg2&& a2)
                        ->      detail::subscribe_factory<decltype  (make_subscriber<T>(std::forward<Arg0>(a0),
std::forward<Arg1>(a1), std::forward<Arg2>(a2)))> {
                        return  detail::subscribe_factory<decltype  (make_subscriber<T>(std::forward<Arg0>(a0),
std::forward<Arg1>(a1), std::forward<Arg2>(a2)))>
                                (make_subscriber<T>(std::forward<Arg0>(a0), std::forward<Arg1>(a1), std::forward<Arg2>(a2)));
                }
                template<class T, class Arg0, class Arg1, class Arg2, class Arg3>
                auto subscribe(Arg0&& a0, Arg1&& a1, Arg2&& a2, Arg3&& a3)
                        ->      detail::subscribe_factory<decltype  (make_subscriber<T>(std::forward<Arg0>(a0),
std::forward<Arg1>(a1), std::forward<Arg2>(a2), std::forward<Arg3>(a3)))> {
                        return  detail::subscribe_factory<decltype  (make_subscriber<T>(std::forward<Arg0>(a0),
std::forward<Arg1>(a1), std::forward<Arg2>(a2), std::forward<Arg3>(a3)))>
                                (make_subscriber<T>(std::forward<Arg0>(a0), std::forward<Arg1>(a1), std::forward<Arg2>(a2),
std::forward<Arg3>(a3)));
                }
                template<class T, class Arg0, class Arg1, class Arg2, class Arg3, class Arg4>
                auto subscribe(Arg0&& a0, Arg1&& a1, Arg2&& a2, Arg3&& a3, Arg4&& a4)
                        ->      detail::subscribe_factory<decltype  (make_subscriber<T>(std::forward<Arg0>(a0),
std::forward<Arg1>(a1), std::forward<Arg2>(a2), std::forward<Arg3>(a3), std::forward<Arg4>(a4)))> {
                        return  detail::subscribe_factory<decltype  (make_subscriber<T>(std::forward<Arg0>(a0),
std::forward<Arg1>(a1), std::forward<Arg2>(a2), std::forward<Arg3>(a3), std::forward<Arg4>(a4)))>
```

```
                                        (make_subscriber<T>(std::forward<Arg0>(a0), std::forward<Arg1>(a1), std::forward<Arg2>(a2),
std::forward<Arg3>(a3), std::forward<Arg4>(a4)));
                        }
                        template<class T, class Arg0, class Arg1, class Arg2, class Arg3, class Arg4, class Arg5>
                        auto subscribe(Arg0&& a0, Arg1&& a1, Arg2&& a2, Arg3&& a3, Arg4&& a4, Arg5&& a5)
                                -> detail::subscribe_factory<decltype (make_subscriber<T>(std::forward<Arg0>(a0),
std::forward<Arg1>(a1), std::forward<Arg2>(a2), std::forward<Arg3>(a3), std::forward<Arg4>(a4), std::forward<Arg5>(a5)))> {
                                return detail::subscribe_factory<decltype (make_subscriber<T>(std::forward<Arg0>(a0),
std::forward<Arg1>(a1), std::forward<Arg2>(a2), std::forward<Arg3>(a3), std::forward<Arg4>(a4), std::forward<Arg5>(a5)))>
                                        (make_subscriber<T>(std::forward<Arg0>(a0), std::forward<Arg1>(a1), std::forward<Arg2>(a2),
std::forward<Arg3>(a3), std::forward<Arg4>(a4), std::forward<Arg5>(a5)));
                        }

                        namespace detail {

                                class dynamic_factory
                                {
                                public:
                                        template<class Observable>
                                        auto operator()(Observable&& source)
                                                -> observable<rxu::value_type_t<rxu::decay_t<Observable>>> {
                                                return
observable<rxu::value_type_t<rxu::decay_t<Observable>>>(std::forward<Observable>(source));
                                        }
                                };

                        }

                        inline auto as_dynamic()
                                -> detail::dynamic_factory {
                                return detail::dynamic_factory();
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_SUBSCRIBE_ON_HPP)
#define RXCPP_OPERATORS_RX_SUBSCRIBE_ON_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Observable, class Coordination>
                        struct subscribe_on : public operator_base<T>
                        {
                                typedef rxu::decay_t<Observable> source_type;
                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef typename coordination_type::coordinator_type coordinator_type;
                                struct subscribe_on_values
                                {
                                        ~subscribe_on_values()
                                        {
                                        }
                                        subscribe_on_values(source_type s, coordination_type sf)
                                                : source(std::move(s))
                                                , coordination(std::move(sf))
                                        {
                                        }
                                        source_type source;
                                        coordination_type coordination;
                                private:
                                        subscribe_on_values& operator=(subscribe_on_values o) RXCPP_DELETE;
                                };
                                const subscribe_on_values initial;

                                ~subscribe_on()
                                {
                                }
                                subscribe_on(source_type s, coordination_type sf)
                                        : initial(std::move(s), std::move(sf))
                                {
                                }
```

```
                template<class Subscriber>
                void on_subscribe(Subscriber s) const {

                        typedef Subscriber output_type;
                        struct subscribe_on_state_type
                                : public std::enable_shared_from_this<subscribe_on_state_type>
                                , public subscribe_on_values
                        {
                                subscribe_on_state_type(const subscribe_on_values& i, const output_type&
oarg)
                                : subscribe_on_values(i)
                                , out(oarg)
                                {
                                }
                                composite_subscription source_lifetime;
                                output_type out;
                        private:
                                subscribe_on_state_type& operator=(subscribe_on_state_type o)
RXCPP_DELETE;
                        };

                        composite_subscription coordinator_lifetime;

                        auto coordinator = initial.coordination.create_coordinator(coordinator_lifetime);

                        auto controller = coordinator.get_worker();

                        // take a copy of the values for each subscription
                        auto state = std::make_shared<subscribe_on_state_type>(initial, std::move(s));

                        auto sl = state->source_lifetime;
                        auto ol = state->out.get_subscription();

                        auto disposer = [=](const rxsc::schedulable&){
                                sl.unsubscribe();
                                ol.unsubscribe();
                                coordinator_lifetime.unsubscribe();
                        };
                        auto selectedDisposer = on_exception(
                                [&](){return coordinator.act(disposer); },
                                state->out);
                        if (selectedDisposer.empty()) {
                                return;
                        }

                        state->source_lifetime.add([=](){
                                controller.schedule(selectedDisposer.get());
                        });

                        state->out.add([=](){
                                sl.unsubscribe();
                                ol.unsubscribe();
                                coordinator_lifetime.unsubscribe();
                        });

                        auto producer = [=](const rxsc::schedulable&){
                                state->source.subscribe(state->source_lifetime, state->out);
                        };

                        auto selectedProducer = on_exception(
                                [&](){return coordinator.act(producer); },
                                state->out);
                        if (selectedProducer.empty()) {
                                return;
                        }

                        controller.schedule(selectedProducer.get());
                }
        private:
                subscribe_on& operator=(subscribe_on o) RXCPP_DELETE;
        };

        template<class Coordination>
        class subscribe_on_factory
        {
                typedef rxu::decay_t<Coordination> coordination_type;

                coordination_type coordination;
        public:
                subscribe_on_factory(coordination_type sf)
                        : coordination(std::move(sf))
                {
```

```
                                        }
                                        template<class Observable>
                                        auto operator()(Observable&& source)
                                                -> observable<rxu::value_type_t<rxu::decay_t<Observable>>,
subscribe_on<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, Coordination>> {
                                                return observable<rxu::value_type_t<rxu::decay_t<Observable>>,
subscribe_on<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, Coordination>>(
                                                        subscribe_on<rxu::value_type_t<rxu::decay_t<Observable>>, Observable,
Coordination>(std::forward<Observable>(source), coordination));
                                        }
                                };

                        }

                        template<class Coordination>
                        auto subscribe_on(Coordination sf)
                                -> detail::subscribe_on_factory<Coordination> {
                                return detail::subscribe_on_factory<Coordination>(std::move(sf));
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_SWITCH_IF_EMPTY_HPP)
#define RXCPP_OPERATORS_RX_SWITCH_IF_EMPTY_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class BackupSource>
                        struct switch_if_empty
                        {
                                typedef rxu::decay_t<T> source_value_type;
                                typedef rxu::decay_t<BackupSource> backup_source_type;

                                backup_source_type backup;

                                switch_if_empty(backup_source_type b)
                                        : backup(std::move(b))
                                {
                                }

                                template<class Subscriber>
                                struct switch_if_empty_observer
                                {
                                        typedef switch_if_empty_observer<Subscriber> this_type;
                                        typedef source_value_type value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<value_type, this_type> observer_type;

                                        dest_type dest;
                                        composite_subscription lifetime;
                                        backup_source_type backup;
                                        mutable bool is_empty;

                                        switch_if_empty_observer(dest_type d, composite_subscription cs, backup_source_type b)
                                                : dest(std::move(d))
                                                , lifetime(std::move(cs))
                                                , backup(std::move(b))
                                                , is_empty(true)
                                        {
                                                dest.add(lifetime);
                                        }
                                        void on_next(source_value_type v) const {
                                                is_empty = false;
                                                dest.on_next(std::move(v));
                                        }
                                        void on_error(std::exception_ptr e) const {
                                                dest.on_error(std::move(e));
                                        }
                                        void on_completed() const {
                                                if (!is_empty) {
                                                        dest.on_completed();
```

```
                                                }
                                                else {
                                                        backup.subscribe(dest);
                                                }
                                        }

                                        static subscriber<value_type, observer_type> make(dest_type d, backup_source_type b) {
                                                auto cs = composite_subscription();
                                                return make_subscriber<value_type>(cs, observer_type(this_type(std::move(d),
cs, std::move(b))));
                                        }
                                };

                                template<class Subscriber>
                                auto operator()(Subscriber dest) const
                                        -> decltype(switch_if_empty_observer<Subscriber>::make(std::move(dest),
std::move(backup))) {
                                        return     switch_if_empty_observer<Subscriber>::make(std::move(dest),
std::move(backup));
                                }
                        };

                        template<class BackupSource>
                        class switch_if_empty_factory
                        {
                                typedef rxu::decay_t<BackupSource> backup_source_type;
                                backup_source_type backup;
                        public:
                                switch_if_empty_factory(backup_source_type b) : backup(std::move(b)) {}
                                template<class Observable>
                                auto operator()(Observable&& source)
                                        -> decltype(source.template
lift<rxu::value_type_t<rxu::decay_t<Observable>>>(switch_if_empty<rxu::value_type_t<rxu::decay_t<Observable>>,
backup_source_type>(backup))) {
                                        return     source.template
lift<rxu::value_type_t<rxu::decay_t<Observable>>>(switch_if_empty<rxu::value_type_t<rxu::decay_t<Observable>>,
backup_source_type>(backup));
                                }
                        };

                }

                template<class BackupSource>
                auto switch_if_empty(BackupSource&& b)
                        ->      detail::switch_if_empty_factory<BackupSource> {
                        return  detail::switch_if_empty_factory<BackupSource>(std::forward<BackupSource>(b));
                }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_SWITCH_ON_NEXT_HPP)
#define RXCPP_OPERATORS_RX_SWITCH_ON_NEXT_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Observable, class Coordination>
                        struct switch_on_next
                                : public operator_base<rxu::value_type_t<rxu::decay_t<T>>>
                        {
                                //static_assert(is_observable<Observable>::value, "switch_on_next requires an observable");
                                //static_assert(is_observable<T>::value, "switch_on_next requires an observable that contains
observables");

                                typedef switch_on_next<T, Observable, Coordination> this_type;

                                typedef rxu::decay_t<T> source_value_type;
                                typedef rxu::decay_t<Observable> source_type;

                                typedef typename source_type::source_operator_type source_operator_type;

                                typedef source_value_type collection_type;
```

```
typedef typename collection_type::value_type collection_value_type;

typedef rxu::decay_t<Coordination> coordination_type;
typedef typename coordination_type::coordinator_type coordinator_type;

struct values
{
	values(source_operator_type o, coordination_type sf)
	: source_operator(std::move(o))
	, coordination(std::move(sf))
	{
	}
	source_operator_type source_operator;
	coordination_type coordination;
};
values initial;

switch_on_next(const source_type& o, coordination_type sf)
	: initial(o.source_operator, std::move(sf))
{
}

template<class Subscriber>
void on_subscribe(Subscriber scbr) const {
	static_assert(is_subscriber<Subscriber>::value, "subscribe must be passed a subscriber");

	typedef Subscriber output_type;

	struct switch_state_type
		: public std::enable_shared_from_this<switch_state_type>
		, public values
	{
		switch_state_type(values i, coordinator_type coor, output_type oarg)
		: values(i)
		, source(i.source_operator)
		, pendingCompletions(0)
		, coordinator(std::move(coor))
		, out(std::move(oarg))
		{
		}
		observable<source_value_type, source_operator_type> source;
		// on_completed on the output must wait until all the
		// subscriptions have received on_completed
		int pendingCompletions;
		coordinator_type coordinator;
		composite_subscription inner_lifetime;
		output_type out;
	};

	auto coordinator = initial.coordination.create_coordinator(scbr.get_subscription());

	// take a copy of the values for each subscription
	auto state = std::make_shared<switch_state_type>(initial, std::move(coordinator),
std::move(scbr));

	composite_subscription outercs;

	// when the out observer is unsubscribed all the
	// inner subscriptions are unsubscribed as well
	state->out.add(outercs);

	auto source = on_exception(
		[&](){return state->coordinator.in(state->source); },
		state->out);
	if (source.empty()) {
		return;
	}

	++state->pendingCompletions;
	// this subscribe does not share the observer subscription
	// so that when it is unsubscribed the observer can be called
	// until the inner subscriptions have finished
	auto sink = make_subscriber<collection_type>(
		state->out,
		outercs,
		// on_next
		[state](collection_type st) {

		state->inner_lifetime.unsubscribe();

		state->inner_lifetime = composite_subscription();
```

```
                                                // when the out observer is unsubscribed all the
                                                // inner subscriptions are unsubscribed as well
                                                auto innerlifetimetoken = state->out.add(state->inner_lifetime);

                                                state->inner_lifetime.add(make_subscription([state, innerlifetimetoken](){
                                                        state->out.remove(innerlifetimetoken);
                                                        --state->pendingCompletions;
                                                }));

                                                auto selectedSource = state->coordinator.in(st);

                                                // this subscribe does not share the source subscription
                                                // so that when it is unsubscribed the source will continue
                                                auto sinkInner = make_subscriber<collection_value_type>(
                                                        state->out,
                                                        state->inner_lifetime,
                                                        // on_next
                                                        [state, st](collection_value_type ct) {
                                                        state->out.on_next(std::move(ct));
                                                },
                                                        // on_error
                                                        [state](std::exception_ptr e) {
                                                        state->out.on_error(e);
                                                },
                                                        //on_completed
                                                        [state](){
                                                        if (state->pendingCompletions == 1) {
                                                                state->out.on_completed();
                                                        }
                                                }
                                                );

                                                auto selectedSinkInner = state->coordinator.out(sinkInner);
                                                ++state->pendingCompletions;
                                                selectedSource.subscribe(std::move(selectedSinkInner));
                                        },
                                                // on_error
                                                [state](std::exception_ptr e) {
                                                state->out.on_error(e);
                                        },
                                                // on_completed
                                                [state]() {
                                                if (--state->pendingCompletions == 0) {
                                                        state->out.on_completed();
                                                }
                                        }
                                        );

                                        auto selectedSink = on_exception(
                                                [&](){return state->coordinator.out(sink); },
                                                state->out);
                                        if (selectedSink.empty()) {
                                                return;
                                        }

                                        source->subscribe(std::move(selectedSink.get()));

                                }
                        };

                        template<class Coordination>
                        class switch_on_next_factory
                        {
                                typedef rxu::decay_t<Coordination> coordination_type;

                                coordination_type coordination;
                        public:
                                switch_on_next_factory(coordination_type sf)
                                        : coordination(std::move(sf))
                                {
                                }

                                template<class Observable>
                                auto operator()(Observable source)
                                        ->      observable<rxu::value_type_t<switch_on_next<rxu::value_type_t<Observable>,
Observable, Coordination>>, switch_on_next<rxu::value_type_t<Observable>, Observable, Coordination>> {
                                        return  observable<rxu::value_type_t<switch_on_next<rxu::value_type_t<Observable>,
Observable, Coordination>>, switch_on_next<rxu::value_type_t<Observable>, Observable, Coordination>>(
                                                switch_on_next<rxu::value_type_t<Observable>, Observable,
Coordination>(std::move(source), coordination));
                                }
                        };
```

```
                }

                template<class Coordination>
                auto switch_on_next(Coordination&& sf)
                        ->      detail::switch_on_next_factory<Coordination> {
                        return  detail::switch_on_next_factory<Coordination>(std::forward<Coordination>(sf));
                }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_TAKE_HPP)
#define RXCPP_OPERATORS_RX_TAKE_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Observable, class Count>
                        struct take : public operator_base<T>
                        {
                                typedef rxu::decay_t<Observable> source_type;
                                typedef rxu::decay_t<Count> count_type;
                                struct values
                                {
                                        values(source_type s, count_type t)
                                        : source(std::move(s))
                                        , count(std::move(t))
                                        {
                                        }
                                        source_type source;
                                        count_type count;
                                };
                                values initial;

                                take(source_type s, count_type t)
                                        : initial(std::move(s), std::move(t))
                                {
                                }

                                struct mode
                                {
                                        enum type {
                                                taking,    // capture messages
                                                triggered, // ignore messages
                                                errored,   // error occured
                                                stopped    // observable completed
                                        };
                                };

                                template<class Subscriber>
                                void on_subscribe(const Subscriber& s) const {

                                        typedef Subscriber output_type;
                                        struct state_type
                                                : public std::enable_shared_from_this<state_type>
                                                , public values
                                        {
                                                state_type(const values& i, const output_type& oarg)
                                                : values(i)
                                                , mode_value(mode::taking)
                                                , out(oarg)
                                                {
                                                }
                                                typename mode::type mode_value;
                                                output_type out;
                                        };
                                        // take a copy of the values for each subscription
                                        auto state = std::make_shared<state_type>(initial, s);

                                        composite_subscription source_lifetime;

                                        s.add(source_lifetime);
```

```
                                        state->source.subscribe(
                                                // split subscription lifetime
                                                source_lifetime,
                                                // on_next
                                                [state, source_lifetime](T t) {
                                                if (state->mode_value < mode::triggered) {
                                                        if (--state->count > 0) {
                                                                state->out.on_next(t);
                                                        }
                                                        else {
                                                                state->mode_value = mode::triggered;
                                                                state->out.on_next(t);
                                                                // must shutdown source before signaling completion
                                                                source_lifetime.unsubscribe();
                                                                state->out.on_completed();
                                                        }
                                                }
                                        },
                                                // on_error
                                                [state](std::exception_ptr e) {
                                                state->mode_value = mode::errored;
                                                state->out.on_error(e);
                                        },
                                                // on_completed
                                                [state]() {
                                                state->mode_value = mode::stopped;
                                                state->out.on_completed();
                                        }
                                        );
                                }
                        };

                        template<class T>
                        class take_factory
                        {
                                typedef rxu::decay_t<T> count_type;
                                count_type count;
                        public:
                                take_factory(count_type t) : count(std::move(t)) {}
                                template<class Observable>
                                auto operator()(Observable&& source)
                                        ->      observable<rxu::value_type_t<rxu::decay_t<Observable>>,
take<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, count_type>> {
                                        return  observable<rxu::value_type_t<rxu::decay_t<Observable>>,
take<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, count_type>>(
                                                take<rxu::value_type_t<rxu::decay_t<Observable>>, Observable,
count_type>(std::forward<Observable>(source), count));
                                }
                        };

                }

                template<class T>
                auto take(T&& t)
                        ->      detail::take_factory<T> {
                        return  detail::take_factory<T>(std::forward<T>(t));
                }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_TAKE_LAST_HPP)
#define RXCPP_OPERATORS_RX_TAKE_LAST_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Observable, class Count>
                        struct take_last : public operator_base<T>
                        {
                                typedef rxu::decay_t<Observable> source_type;
                                typedef rxu::decay_t<Count> count_type;
```

```
		typedef std::queue<T> queue_type;
		typedef typename queue_type::size_type queue_size_type;

		struct values
		{
			values(source_type s, count_type t)
			: source(std::move(s))
			, count(static_cast<queue_size_type>(t))
			{
			}
			source_type source;
			queue_size_type count;
		};
		values initial;

		take_last(source_type s, count_type t)
			: initial(std::move(s), std::move(t))
		{
		}

		template<class Subscriber>
		void on_subscribe(const Subscriber& s) const {

			typedef Subscriber output_type;
			struct state_type
				: public std::enable_shared_from_this<state_type>
				, public values
			{
				state_type(const values& i, const output_type& oarg)
				: values(i)
				, out(oarg)
				{
				}
				queue_type items;
				output_type out;
			};
			// take a copy of the values for each subscription
			auto state = std::make_shared<state_type>(initial, s);

			composite_subscription source_lifetime;

			s.add(source_lifetime);

			state->source.subscribe(
				// split subscription lifetime
				source_lifetime,
				// on_next
				[state, source_lifetime](T t) {
				if (state->count > 0) {
					if (state->items.size() == state->count) {
						state->items.pop();
					}
					state->items.push(t);
				}
			},
				// on_error
				[state](std::exception_ptr e) {
				state->out.on_error(e);
			},
				// on_completed
				[state]() {
				while (!state->items.empty()) {
					state->out.on_next(std::move(state->items.front()));
					state->items.pop();
				}
				state->out.on_completed();
			}
			);
		}
};

template<class T>
class take_last_factory
{
		typedef rxu::decay_t<T> count_type;
		count_type count;
public:
		take_last_factory(count_type t) : count(std::move(t)) {}
		template<class Observable>
		auto operator()(Observable&& source)
```

```
                                        ->      observable<rxu::value_type_t<rxu::decay_t<Observable>>,
take_last<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, count_type>> {
                                        return  observable<rxu::value_type_t<rxu::decay_t<Observable>>,
take_last<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, count_type>>(
                                                take_last<rxu::value_type_t<rxu::decay_t<Observable>>, Observable,
count_type>(std::forward<Observable>(source), count));
                                        }
                                };

                        }

                        template<class T>
                        auto take_last(T&& t)
                                ->      detail::take_last_factory<T> {
                                return  detail::take_last_factory<T>(std::forward<T>(t));
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_TAKE_UNTIL_HPP)
#define RXCPP_OPERATORS_RX_TAKE_UNTIL_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Observable, class TriggerObservable, class Coordination>
                        struct take_until : public operator_base<T>
                        {
                                typedef rxu::decay_t<Observable> source_type;
                                typedef rxu::decay_t<TriggerObservable> trigger_source_type;
                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef typename coordination_type::coordinator_type coordinator_type;
                                struct values
                                {
                                        values(source_type s, trigger_source_type t, coordination_type sf)
                                        : source(std::move(s))
                                        , trigger(std::move(t))
                                        , coordination(std::move(sf))
                                        {
                                        }
                                        source_type source;
                                        trigger_source_type trigger;
                                        coordination_type coordination;
                                };
                                values initial;

                                take_until(source_type s, trigger_source_type t, coordination_type sf)
                                        : initial(std::move(s), std::move(t), std::move(sf))
                                {
                                }

                                struct mode
                                {
                                        enum type {
                                                taking,    // no messages from trigger
                                                clear,     // trigger completed
                                                triggered, // trigger sent on_next
                                                errored,   // error either on trigger or on observable
                                                stopped    // observable completed
                                        };
                                };

                                template<class Subscriber>
                                void on_subscribe(Subscriber s) const {

                                        typedef Subscriber output_type;
                                        struct take_until_state_type
                                                : public std::enable_shared_from_this<take_until_state_type>
                                                , public values
                                        {
                                                take_until_state_type(const values& i, coordinator_type coor, const
output_type& oarg)
```

```
                : values(i)
                , mode_value(mode::taking)
                , coordinator(std::move(coor))
                , out(oarg)
                {
                        out.add(trigger_lifetime);
                        out.add(source_lifetime);
                }
                typename mode::type mode_value;
                composite_subscription trigger_lifetime;
                composite_subscription source_lifetime;
                coordinator_type coordinator;
                output_type out;
};

auto coordinator = initial.coordination.create_coordinator(s.get_subscription());

// take a copy of the values for each subscription
auto state = std::make_shared<take_until_state_type>(initial, std::move(coordinator),
std::move(s));

auto trigger = on_exception(
        [&](){return state->coordinator.in(state->trigger); },
        state->out);
if (trigger.empty()) {
        return;
}

auto source = on_exception(
        [&](){return state->coordinator.in(state->source); },
        state->out);
if (source.empty()) {
        return;
}

auto sinkTrigger = make_subscriber<typename trigger_source_type::value_type>(
        // share parts of subscription
        state->out,
        // new lifetime
        state->trigger_lifetime,
        // on_next
        [state](const typename trigger_source_type::value_type&) {
        if (state->mode_value != mode::taking) { return; }
        state->mode_value = mode::triggered;
        state->out.on_completed();
},
        // on_error
        [state](std::exception_ptr e) {
        if (state->mode_value != mode::taking) { return; }
        state->mode_value = mode::errored;
        state->out.on_error(e);
},
        // on_completed
        [state]() {
        if (state->mode_value != mode::taking) { return; }
        state->mode_value = mode::clear;
}
);
auto selectedSinkTrigger = on_exception(
        [&](){return state->coordinator.out(sinkTrigger); },
        state->out);
if (selectedSinkTrigger.empty()) {
        return;
}
trigger->subscribe(std::move(selectedSinkTrigger.get()));

auto sinkSource = make_subscriber<T>(
        // split subscription lifetime
        state->source_lifetime,
        // on_next
        [state](T t) {
        //
        // everything is crafted to minimize the overhead of this function.
        //
        if (state->mode_value < mode::triggered) {
                state->out.on_next(t);
        }
},
        // on_error
        [state](std::exception_ptr e) {
        if (state->mode_value > mode::clear) { return; }
        state->mode_value = mode::errored;
```

```
                                                state->out.on_error(e);
                                        },
                                                // on_completed
                                                [state]() {
                                                if (state->mode_value > mode::clear) { return; }
                                                state->mode_value = mode::stopped;
                                                state->out.on_completed();
                                        }
                                        );
                                        auto selectedSinkSource = on_exception(
                                                [&](){return state->coordinator.out(sinkSource); },
                                                state->out);
                                        if (selectedSinkSource.empty()) {
                                                return;
                                        }
                                        source->subscribe(std::move(selectedSinkSource.get()));
                                }
                        };

                        template<class TriggerObservable, class Coordination>
                        class take_until_factory
                        {
                                typedef rxu::decay_t<TriggerObservable> trigger_source_type;
                                typedef rxu::decay_t<Coordination> coordination_type;

                                trigger_source_type trigger_source;
                                coordination_type coordination;
                        public:
                                take_until_factory(trigger_source_type t, coordination_type sf)
                                        : trigger_source(std::move(t))
                                        , coordination(std::move(sf))
                                {
                                }
                                template<class Observable>
                                auto operator()(Observable&& source)
                                        ->      observable<rxu::value_type_t<rxu::decay_t<Observable>>,
take_until<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, trigger_source_type, Coordination>> {
                                        return  observable<rxu::value_type_t<rxu::decay_t<Observable>>,
take_until<rxu::value_type_t<rxu::decay_t<Observable>>, Observable, trigger_source_type, Coordination>>(
                                                take_until<rxu::value_type_t<rxu::decay_t<Observable>>, Observable,
trigger_source_type, Coordination>(std::forward<Observable>(source), trigger_source, coordination));
                                }
                        };

                }

                template<class TriggerObservable, class Coordination>
                auto take_until(TriggerObservable t, Coordination sf)
                        ->      detail::take_until_factory<TriggerObservable, Coordination> {
                        return  detail::take_until_factory<TriggerObservable, Coordination>(std::move(t), std::move(sf));
                }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_TAP_HPP)
#define RXCPP_OPERATORS_RX_TAP_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class MakeObserverArgN>
                        struct tap_observer_factory;

                        template<class T, class... ArgN>
                        struct tap_observer_factory<T, std::tuple<ArgN...>>
                        {
                                using source_value_type = rxu::decay_t<T>;
                                using out_type = decltype(make_observer<source_value_type,
rxcpp::detail::OnErrorIgnore>(*((ArgN*)nullptr)...));
                                auto operator()(ArgN&&... an) -> out_type const {
                                        return make_observer<source_value_type,
rxcpp::detail::OnErrorIgnore>(std::forward<ArgN>(an)...);
```

```
                }
        };

        template<class T, class MakeObserverArgN, class Factory = tap_observer_factory<T, MakeObserverArgN>>
        struct tap
        {
                using source_value_type = rxu::decay_t<T>;
                using args_type = rxu::decay_t<MakeObserverArgN>;
                using factory_type = Factory;
                using out_type = typename factory_type::out_type;
                out_type out;

                tap(args_type a)
                        : out(rxu::apply(std::move(a), factory_type()))
                {
                }

                template<class Subscriber>
                struct tap_observer
                {
                        using this_type = tap_observer<Subscriber>;
                        using value_type = source_value_type;
                        using dest_type = rxu::decay_t<Subscriber>;
                        using factory_type = Factory;
                        using out_type = typename factory_type::out_type;
                        using observer_type = observer<value_type, this_type>;
                        dest_type dest;
                        out_type out;

                        tap_observer(dest_type d, out_type o)
                                : dest(std::move(d))
                                , out(std::move(o))
                        {
                        }
                        void on_next(source_value_type v) const {
                                out.on_next(v);
                                dest.on_next(v);
                        }
                        void on_error(std::exception_ptr e) const {
                                out.on_error(e);
                                dest.on_error(e);
                        }
                        void on_completed() const {
                                out.on_completed();
                                dest.on_completed();
                        }

                        static subscriber<value_type, observer<value_type, this_type>> make(dest_type d,
out_type o) {
                                return make_subscriber<value_type>(d, this_type(d, std::move(o)));
                        }
                };

                template<class Subscriber>
                auto operator()(Subscriber dest) const
                        -> decltype(tap_observer<Subscriber>::make(std::move(dest), out)) {
                        return     tap_observer<Subscriber>::make(std::move(dest), out);
                }
        };

        template<class MakeObserverArgN>
        class tap_factory
        {
                typedef rxu::decay_t<MakeObserverArgN> args_type;
                args_type args;
        public:
                tap_factory(args_type a) : args(std::move(a)) {}
                template<class Observable>
                auto operator()(Observable&& source)
                        -> decltype(source.template
lift<rxu::value_type_t<rxu::decay_t<Observable>>>(tap<rxu::value_type_t<rxu::decay_t<Observable>>, args_type>(args))) {
                        return     source.template
lift<rxu::value_type_t<rxu::decay_t<Observable>>>(tap<rxu::value_type_t<rxu::decay_t<Observable>>, args_type>(args));
                }
        };

    }

    template<class... MakeObserverArgN>
    auto tap(MakeObserverArgN&&... an)
        ->     detail::tap_factory<std::tuple<rxu::decay_t<MakeObserverArgN>...>> {
```

```
                                        return
detail::tap_factory<std::tuple<rxu::decay_t<MakeObserverArgN>...>>(std::make_tuple(std::forward<MakeObserverArgN>(an)...));
                                }

                        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_TIME_INTERVAL_HPP)
#define RXCPP_OPERATORS_RX_TIME_INTERVAL_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Coordination>
                        struct time_interval
                        {
                                static_assert(is_coordination<Coordination>::value, "Coordination parameter must satisfy the
requirements for a Coordination");

                                typedef rxu::decay_t<T> source_value_type;
                                typedef rxu::decay_t<Coordination> coordination_type;

                                struct time_interval_values {
                                        time_interval_values(coordination_type c)
                                        : coordination(c)
                                        {
                                        }

                                        coordination_type coordination;
                                };
                                time_interval_values initial;

                                time_interval(coordination_type coordination)
                                        : initial(coordination)
                                {
                                }

                                template<class Subscriber>
                                struct time_interval_observer
                                {
                                        typedef time_interval_observer<Subscriber> this_type;
                                        typedef source_value_type value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<value_type, this_type> observer_type;
                                        typedef rxsc::scheduler::clock_type::time_point time_point;
                                        dest_type dest;
                                        coordination_type coord;
                                        mutable time_point last;

                                        time_interval_observer(dest_type d, coordination_type coordination)
                                                : dest(std::move(d)),
                                                coord(std::move(coordination)),
                                                last(coord.now())
                                        {
                                        }

                                        void on_next(source_value_type) const {
                                                time_point now = coord.now();
                                                dest.on_next(now - last);
                                                last = now;
                                        }
                                        void on_error(std::exception_ptr e) const {
                                                dest.on_error(e);
                                        }
                                        void on_completed() const {
                                                dest.on_completed();
                                        }

                                        static subscriber<value_type, observer<value_type, this_type>> make(dest_type d,
time_interval_values v) {
                                                return make_subscriber<value_type>(d, this_type(d, v.coordination));
                                        }
                                };
```

```
                                        template<class Subscriber>
                                        auto operator()(Subscriber dest) const
                                                -> decltype(time_interval_observer<Subscriber>::make(std::move(dest), initial)) {
                                                return      time_interval_observer<Subscriber>::make(std::move(dest), initial);
                                        }
                                };

                                template <class Coordination>
                                class time_interval_factory
                                {
                                        typedef rxu::decay_t<Coordination> coordination_type;

                                        coordination_type coordination;
                                public:
                                        time_interval_factory(coordination_type ct)
                                                : coordination(std::move(ct)) { }

                                        template<class Observable>
                                        auto operator()(Observable&& source)
                                                -> decltype(source.template lift<typename
rxsc::scheduler::clock_type::time_point::duration>(time_interval<rxu::value_type_t<rxu::decay_t<Observable>>, Coordination>(coordination)))
{
                                                return      source.template lift<typename
rxsc::scheduler::clock_type::time_point::duration>(time_interval<rxu::value_type_t<rxu::decay_t<Observable>>, Coordination>(coordination));
                                        }

                                };

                        }

                        template <class Coordination>
                        inline auto time_interval(Coordination ct)
                                ->      detail::time_interval_factory<Coordination> {
                                return  detail::time_interval_factory<Coordination>(std::move(ct));
                        }

                        inline auto time_interval()
                                ->      detail::time_interval_factory<identity_one_worker> {
                                return  detail::time_interval_factory<identity_one_worker>(identity_current_thread());
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_TIMEOUT_HPP)
#define RXCPP_OPERATORS_RX_TIMEOUT_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        class timeout_error : public std::runtime_error
        {
        public:
                explicit timeout_error(const std::string& msg) :
                        std::runtime_error(msg)
                {}
        };

        namespace operators {

                namespace detail {

                        template<class T, class Duration, class Coordination>
                        struct timeout
                        {
                                typedef rxu::decay_t<T> source_value_type;
                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef typename coordination_type::coordinator_type coordinator_type;
                                typedef rxu::decay_t<Duration> duration_type;

                                struct timeout_values
                                {
                                        timeout_values(duration_type p, coordination_type c)
                                        : period(p)
                                        , coordination(c)
                                        {
```

```
                }

                duration_type period;
                coordination_type coordination;
        };
        timeout_values initial;

        timeout(duration_type period, coordination_type coordination)
                : initial(period, coordination)
        {
        }

        template<class Subscriber>
        struct timeout_observer
        {
                typedef timeout_observer<Subscriber> this_type;
                typedef rxu::decay_t<T> value_type;
                typedef rxu::decay_t<Subscriber> dest_type;
                typedef observer<T, this_type> observer_type;

                struct timeout_subscriber_values : public timeout_values
                {
                        timeout_subscriber_values(composite_subscription cs, dest_type d,
timeout_values v, coordinator_type c)
                        : timeout_values(v)
                        , cs(std::move(cs))
                        , dest(std::move(d))
                        , coordinator(std::move(c))
                        , worker(coordinator.get_worker())
                        , index(0)
                        {
                        }

                        composite_subscription cs;
                        dest_type dest;
                        coordinator_type coordinator;
                        rxsc::worker worker;
                        mutable std::size_t index;
                };
                typedef std::shared_ptr<timeout_subscriber_values> state_type;
                state_type state;

                timeout_observer(composite_subscription cs, dest_type d, timeout_values v,
coordinator_type c)
                        :
state(std::make_shared<timeout_subscriber_values>(timeout_subscriber_values(std::move(cs), std::move(d), v, std::move(c))))
                {
                        auto localState = state;

                        auto disposer = [=](const rxsc::schedulable&){
                                localState->cs.unsubscribe();
                                localState->dest.unsubscribe();
                                localState->worker.unsubscribe();
                        };
                        auto selectedDisposer = on_exception(
                                [&](){ return localState->coordinator.act(disposer); },
                                localState->dest);
                        if (selectedDisposer.empty()) {
                                return;
                        }

                        localState->dest.add([=](){
                                localState->worker.schedule(selectedDisposer.get());
                        });
                        localState->cs.add([=](){
                                localState->worker.schedule(selectedDisposer.get());
                        });
                }

                static std::function<void(const rxsc::schedulable&)> produce_timeout(std::size_t id,
state_type state) {
                        auto produce = [id, state](const rxsc::schedulable&) {
                                if (id != state->index)
                                        return;

                                state-
>dest.on_error(std::make_exception_ptr(rxcpp::timeout_error("timeout has occurred")));
                        };

                        auto selectedProduce = on_exception(
                                [&](){ return state->coordinator.act(produce); },
                                state->dest);
```

```
                                                if (selectedProduce.empty()) {
                                                        return std::function<void(const rxsc::schedulable&)>();
                                                }

                                                return std::function<void(const rxsc::schedulable&)>(selectedProduce.get());
                                        }

                                        void on_next(T v) const {
                                                auto localState = state;
                                                auto work = [v, localState](const rxsc::schedulable&) {
                                                        auto new_id = ++localState->index;
                                                        auto produce_time = localState->worker.now() + localState->period;

                                                        localState->dest.on_next(v);
                                                        localState->worker.schedule(produce_time,
produce_timeout(new_id, localState));
                                                };
                                                auto selectedWork = on_exception(
                                                        [&](){return localState->coordinator.act(work); },
                                                        localState->dest);
                                                if (selectedWork.empty()) {
                                                        return;
                                                }
                                                localState->worker.schedule(selectedWork.get());
                                        }

                                        void on_error(std::exception_ptr e) const {
                                                auto localState = state;
                                                auto work = [e, localState](const rxsc::schedulable&) {
                                                        localState->dest.on_error(e);
                                                };
                                                auto selectedWork = on_exception(
                                                        [&](){ return localState->coordinator.act(work); },
                                                        localState->dest);
                                                if (selectedWork.empty()) {
                                                        return;
                                                }
                                                localState->worker.schedule(selectedWork.get());
                                        }

                                        void on_completed() const {
                                                auto localState = state;
                                                auto work = [localState](const rxsc::schedulable&) {
                                                        localState->dest.on_completed();
                                                };
                                                auto selectedWork = on_exception(
                                                        [&](){ return localState->coordinator.act(work); },
                                                        localState->dest);
                                                if (selectedWork.empty()) {
                                                        return;
                                                }
                                                localState->worker.schedule(selectedWork.get());
                                        }

                                        static subscriber<T, observer_type> make(dest_type d, timeout_values v) {
                                                auto cs = composite_subscription();
                                                auto coordinator = v.coordination.create_coordinator();

                                                return make_subscriber<T>(cs, observer_type(this_type(cs, std::move(d),
std::move(v), std::move(coordinator))));
                                        }
                                };

                                template<class Subscriber>
                                auto operator()(Subscriber dest) const
                                        -> decltype(timeout_observer<Subscriber>::make(std::move(dest), initial)) {
                                        return      timeout_observer<Subscriber>::make(std::move(dest), initial);
                                }
                        };

                        template<class Duration, class Coordination>
                        class timeout_factory
                        {
                                typedef rxu::decay_t<Duration> duration_type;
                                typedef rxu::decay_t<Coordination> coordination_type;

                                duration_type period;
                                coordination_type coordination;
                        public:
                                timeout_factory(duration_type p, coordination_type c) : period(p), coordination(c) {}
                                template<class Observable>
                                auto operator()(Observable&& source)
```

```
                                                -> decltype(source.template
lift<rxu::value_type_t<rxu::decay_t<Observable>>>(timeout<rxu::value_type_t<rxu::decay_t<Observable>>, Duration, Coordination>(period,
coordination))) {
                                                return      source.template
lift<rxu::value_type_t<rxu::decay_t<Observable>>>(timeout<rxu::value_type_t<rxu::decay_t<Observable>>, Duration, Coordination>(period,
coordination));
                                        }
                                };

                        }

                        template<class Duration, class Coordination>
                        inline auto timeout(Duration period, Coordination coordination)
                                ->      detail::timeout_factory<Duration, Coordination> {
                                return  detail::timeout_factory<Duration, Coordination>(period, coordination);
                        }

                        template<class Duration>
                        inline auto timeout(Duration period)
                                ->      detail::timeout_factory<Duration, identity_one_worker> {
                                return  detail::timeout_factory<Duration, identity_one_worker>(period, identity_current_thread());
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_TIMESTAMP_HPP)
#define RXCPP_OPERATORS_RX_TIMESTAMP_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Coordination>
                        struct timestamp
                        {
                                static_assert(is_coordination<Coordination>::value, "Coordination parameter must satisfy the
requirements for a Coordination");

                                typedef rxu::decay_t<T> source_value_type;
                                typedef rxu::decay_t<Coordination> coordination_type;

                                struct timestamp_values {
                                        timestamp_values(coordination_type c)
                                        : coordination(c)
                                        {
                                        }

                                        coordination_type coordination;
                                };
                                timestamp_values initial;

                                timestamp(coordination_type coordination)
                                        : initial(coordination)
                                {
                                }

                                template<class Subscriber>
                                struct timestamp_observer
                                {
                                        typedef timestamp_observer<Subscriber> this_type;
                                        typedef source_value_type value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<value_type, this_type> observer_type;
                                        dest_type dest;
                                        coordination_type coord;

                                        timestamp_observer(dest_type d, coordination_type coordination)
                                                : dest(std::move(d)),
                                                coord(std::move(coordination))
                                        {
                                        }

                                        void on_next(source_value_type v) const {
```

```
                                                dest.on_next(std::make_pair(v, coord.now()));
                                        }
                                        void on_error(std::exception_ptr e) const {
                                                dest.on_error(e);
                                        }
                                        void on_completed() const {
                                                dest.on_completed();
                                        }

                                        static subscriber<value_type, observer<value_type, this_type>> make(dest_type d,
timestamp_values v) {
                                                return make_subscriber<value_type>(d, this_type(d, v.coordination));
                                        }
                                };

                                template<class Subscriber>
                                auto operator()(Subscriber dest) const
                                        -> decltype(timestamp_observer<Subscriber>::make(std::move(dest), initial)) {
                                        return      timestamp_observer<Subscriber>::make(std::move(dest), initial);
                                }
                        };

                        template <class Coordination>
                        class timestamp_factory
                        {
                                typedef rxu::decay_t<Coordination> coordination_type;

                                coordination_type coordination;
                        public:
                                timestamp_factory(coordination_type ct)
                                        : coordination(std::move(ct)) { }

                                template<class Observable>
                                auto operator()(Observable&& source)
                                        -> decltype(source.template lift<std::pair<rxu::value_type_t<rxu::decay_t<Observable>>,
typename rxsc::scheduler::clock_type::time_point>>(timestamp<rxu::value_type_t<rxu::decay_t<Observable>>, Coordination>(coordination))) {
                                        return      source.template lift<std::pair<rxu::value_type_t<rxu::decay_t<Observable>>,
typename rxsc::scheduler::clock_type::time_point>>(timestamp<rxu::value_type_t<rxu::decay_t<Observable>>, Coordination>(coordination));
                                }

                        };

                }

                template <class Coordination>
                inline auto timestamp(Coordination ct)
                        ->      detail::timestamp_factory<Coordination> {
                        return  detail::timestamp_factory<Coordination>(std::move(ct));
                }

                inline auto timestamp()
                        ->      detail::timestamp_factory<identity_one_worker> {
                        return  detail::timestamp_factory<identity_one_worker>(identity_current_thread());
                }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_WINDOW_HPP)
#define RXCPP_OPERATORS_RX_WINDOW_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T>
                        struct window
                        {
                                typedef rxu::decay_t<T> source_value_type;
                                struct window_values
                                {
                                        window_values(int c, int s)
                                        : count(c)
                                        , skip(s)
```

```
                {
                }
                int count;
                int skip;
            };

            window_values initial;

            window(int count, int skip)
                : initial(count, skip)
            {
            }

            template<class Subscriber>
            struct window_observer : public window_values
            {
                typedef window_observer<Subscriber> this_type;
                typedef rxu::decay_t<T> value_type;
                typedef rxu::decay_t<Subscriber> dest_type;
                typedef observer<T, this_type> observer_type;
                dest_type dest;
                mutable int cursor;
                mutable std::deque<rxcpp::subjects::subject<T>> subj;

                window_observer(dest_type d, window_values v)
                    : window_values(v)
                    , dest(std::move(d))
                    , cursor(0)
                {
                    subj.push_back(rxcpp::subjects::subject<T>());
                    dest.on_next(subj[0].get_observable().as_dynamic());
                }
                void on_next(T v) const {
                    for (auto s : subj) {
                        s.get_subscriber().on_next(v);
                    }

                    int c = cursor - this->count + 1;
                    if (c >= 0 && c % this->skip == 0) {
                        subj[0].get_subscriber().on_completed();
                        subj.pop_front();
                    }

                    if (++cursor % this->skip == 0) {
                        subj.push_back(rxcpp::subjects::subject<T>());
                        dest.on_next(subj[subj.size() - 1].get_observable().as_dynamic());
                    }
                }

                void on_error(std::exception_ptr e) const {
                    for (auto s : subj) {
                        s.get_subscriber().on_error(e);
                    }
                    dest.on_error(e);
                }

                void on_completed() const {
                    for (auto s : subj) {
                        s.get_subscriber().on_completed();
                    }
                    dest.on_completed();
                }

                static subscriber<T, observer_type> make(dest_type d, window_values v) {
                    auto cs = d.get_subscription();
                    return make_subscriber<T>(std::move(cs),
observer_type(this_type(std::move(d), std::move(v))));
                }
            };

            template<class Subscriber>
            auto operator()(Subscriber dest) const
                -> decltype(window_observer<Subscriber>::make(std::move(dest), initial)) {
                return      window_observer<Subscriber>::make(std::move(dest), initial);
            }
        };

        class window_factory
        {
            int count;
            int skip;
        public:
```

```
                                        window_factory(int c, int s) : count(c), skip(s) {}
                                        template<class Observable>
                                        auto operator()(Observable&& source)
                                                -> decltype(source.template
lift<observable<rxu::value_type_t<rxu::decay_t<Observable>>>>(window<rxu::value_type_t<rxu::decay_t<Observable>>>(count, skip))) {
                                                return      source.template
lift<observable<rxu::value_type_t<rxu::decay_t<Observable>>>>(window<rxu::value_type_t<rxu::decay_t<Observable>>>(count, skip));
                                        }
                                };

                        }

                        inline auto window(int count)
                                ->      detail::window_factory {
                                return  detail::window_factory(count, count);
                        }
                        inline auto window(int count, int skip)
                                ->      detail::window_factory {
                                return  detail::window_factory(count, skip);
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_WINDOW_WITH_TIME_HPP)
#define RXCPP_OPERATORS_RX_WINDOW_WITH_TIME_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Duration, class Coordination>
                        struct window_with_time
                        {
                                typedef rxu::decay_t<T> source_value_type;
                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef typename coordination_type::coordinator_type coordinator_type;
                                typedef rxu::decay_t<Duration> duration_type;

                                struct window_with_time_values
                                {
                                        window_with_time_values(duration_type p, duration_type s, coordination_type c)
                                        : period(p)
                                        , skip(s)
                                        , coordination(c)
                                        {
                                        }
                                        duration_type period;
                                        duration_type skip;
                                        coordination_type coordination;
                                };
                                window_with_time_values initial;

                                window_with_time(duration_type period, duration_type skip, coordination_type coordination)
                                        : initial(period, skip, coordination)
                                {
                                }

                                template<class Subscriber>
                                struct window_with_time_observer
                                {
                                        typedef window_with_time_observer<Subscriber> this_type;
                                        typedef rxu::decay_t<T> value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<T, this_type> observer_type;

                                        struct window_with_time_subscriber_values : public window_with_time_values
                                        {
                                                window_with_time_subscriber_values(composite_subscription cs, dest_type d,
window_with_time_values v, coordinator_type c)
                                                : window_with_time_values(v)
                                                , cs(std::move(cs))
                                                , dest(std::move(d))
                                                , coordinator(std::move(c))
```

```
                , worker(coordinator.get_worker())
                , expected(worker.now())
                {
                }
                composite_subscription cs;
                dest_type dest;
                coordinator_type coordinator;
                rxsc::worker worker;
                mutable std::deque<rxcpp::subjects::subject<T>> subj;
                rxsc::scheduler::clock_type::time_point expected;
        };
        std::shared_ptr<window_with_time_subscriber_values> state;

        window_with_time_observer(composite_subscription cs, dest_type d,
window_with_time_values v, coordinator_type c)
                :
state(std::make_shared<window_with_time_subscriber_values>(window_with_time_subscriber_values(std::move(cs), std::move(d), v,
std::move(c))))
        {
                auto localState = state;

                auto disposer = [=](const rxsc::schedulable&){
                        localState->cs.unsubscribe();
                        localState->dest.unsubscribe();
                        localState->worker.unsubscribe();
                };
                auto selectedDisposer = on_exception(
                        [&](){return localState->coordinator.act(disposer); },
                        localState->dest);
                if (selectedDisposer.empty()) {
                        return;
                }

                localState->dest.add([=](){
                        localState->worker.schedule(selectedDisposer.get());
                });
                localState->cs.add([=](){
                        localState->worker.schedule(selectedDisposer.get());
                });

                //
                // The scheduler is FIFO for any time T. Since the observer is scheduling
                // on_next/on_error/oncompleted the timed schedule calls must be resheduled
                // when they occur to ensure that production happens after
on_next/on_error/oncompleted
                //

                auto release_window = [localState](const rxsc::schedulable&) {
                        localState->worker.schedule([localState](const rxsc::schedulable&) {
                                localState->subj[0].get_subscriber().on_completed();
                                localState->subj.pop_front();
                        });
                };
                auto selectedRelease = on_exception(
                        [&](){return localState->coordinator.act(release_window); },
                        localState->dest);
                if (selectedRelease.empty()) {
                        return;
                }

                auto create_window = [localState, selectedRelease](const rxsc::schedulable&) {
                        localState->subj.push_back(rxcpp::subjects::subject<T>());
                        localState->dest.on_next(localState->subj[localState->subj.size() -
1].get_observable().as_dynamic());

                        auto produce_at = localState->expected + localState->period;
                        localState->expected += localState->skip;
                        localState->worker.schedule(produce_at, [localState,
selectedRelease](const rxsc::schedulable&) {
                                localState->worker.schedule(selectedRelease.get());
                        });
                };
                auto selectedCreate = on_exception(
                        [&](){return localState->coordinator.act(create_window); },
                        localState->dest);
                if (selectedCreate.empty()) {
                        return;
                }

                state->worker.schedule_periodically(
                        state->expected,
                        state->skip,
```

```
                                                [localState, selectedCreate](const rxsc::schedulable&) {
                                                localState->worker.schedule(selectedCreate.get());
                                        });
                                }

                                void on_next(T v) const {
                                        auto localState = state;
                                        auto work = [v, localState](const rxsc::schedulable&){
                                                for (auto s : localState->subj) {
                                                        s.get_subscriber().on_next(v);
                                                }
                                        };
                                        auto selectedWork = on_exception(
                                                [&](){return localState->coordinator.act(work); },
                                                localState->dest);
                                        if (selectedWork.empty()) {
                                                return;
                                        }
                                        localState->worker.schedule(selectedWork.get());
                                }

                                void on_error(std::exception_ptr e) const {
                                        auto localState = state;
                                        auto work = [e, localState](const rxsc::schedulable&){
                                                for (auto s : localState->subj) {
                                                        s.get_subscriber().on_error(e);
                                                }
                                                localState->dest.on_error(e);
                                        };
                                        auto selectedWork = on_exception(
                                                [&](){return localState->coordinator.act(work); },
                                                localState->dest);
                                        if (selectedWork.empty()) {
                                                return;
                                        }
                                        localState->worker.schedule(selectedWork.get());
                                }

                                void on_completed() const {
                                        auto localState = state;
                                        auto work = [localState](const rxsc::schedulable&){
                                                for (auto s : localState->subj) {
                                                        s.get_subscriber().on_completed();
                                                }
                                                localState->dest.on_completed();
                                        };
                                        auto selectedWork = on_exception(
                                                [&](){return localState->coordinator.act(work); },
                                                localState->dest);
                                        if (selectedWork.empty()) {
                                                return;
                                        }
                                        localState->worker.schedule(selectedWork.get());
                                }

                                static subscriber<T, observer_type> make(dest_type d, window_with_time_values v) {
                                        auto cs = composite_subscription();
                                        auto coordinator = v.coordination.create_coordinator();

                                        return make_subscriber<T>(cs, observer_type(this_type(cs, std::move(d),
std::move(v), std::move(coordinator))));
                                }
                        };

                        template<class Subscriber>
                        auto operator()(Subscriber dest) const
                                -> decltype(window_with_time_observer<Subscriber>::make(std::move(dest), initial)) {
                                return     window_with_time_observer<Subscriber>::make(std::move(dest), initial);
                        }
                };

                template<class Duration, class Coordination>
                class window_with_time_factory
                {
                        typedef rxu::decay_t<Duration> duration_type;
                        typedef rxu::decay_t<Coordination> coordination_type;

                        duration_type period;
                        duration_type skip;
                        coordination_type coordination;
                public:
```

```
                                        window_with_time_factory(duration_type p, duration_type s, coordination_type c) : period(p),
skip(s), coordination(c) {}
                                        template<class Observable>
                                        auto operator()(Observable&& source)
                                                -> decltype(source.template
lift<observable<rxu::value_type_t<rxu::decay_t<Observable>>>>(window_with_time<rxu::value_type_t<rxu::decay_t<Observable>>, Duration,
Coordination>(period, skip, coordination))) {
                                                return     source.template
lift<observable<rxu::value_type_t<rxu::decay_t<Observable>>>>(window_with_time<rxu::value_type_t<rxu::decay_t<Observable>>, Duration,
Coordination>(period, skip, coordination));
                                        }
                                };

                        }

                        template<class Duration, class Coordination>
                        inline auto window_with_time(Duration period, Coordination coordination)
                                ->      detail::window_with_time_factory<Duration, Coordination> {
                                return  detail::window_with_time_factory<Duration, Coordination>(period, period, coordination);
                        }

                        template<class Duration, class Coordination>
                        inline auto window_with_time(Duration period, Duration skip, Coordination coordination)
                                ->      detail::window_with_time_factory<Duration, Coordination> {
                                return  detail::window_with_time_factory<Duration, Coordination>(period, skip, coordination);
                        }

                }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_WINDOW_WITH_TIME_OR_COUNT_HPP)
#define RXCPP_OPERATORS_RX_WINDOW_WITH_TIME_OR_COUNT_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Duration, class Coordination>
                        struct window_with_time_or_count
                        {
                                typedef rxu::decay_t<T> source_value_type;
                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef typename coordination_type::coordinator_type coordinator_type;
                                typedef rxu::decay_t<Duration> duration_type;

                                struct window_with_time_or_count_values
                                {
                                        window_with_time_or_count_values(duration_type p, int n, coordination_type c)
                                        : period(p)
                                        , count(n)
                                        , coordination(c)
                                        {
                                        }
                                        duration_type period;
                                        int count;
                                        coordination_type coordination;
                                };
                                window_with_time_or_count_values initial;

                                window_with_time_or_count(duration_type period, int count, coordination_type coordination)
                                        : initial(period, count, coordination)
                                {
                                }

                                template<class Subscriber>
                                struct window_with_time_or_count_observer
                                {
                                        typedef window_with_time_or_count_observer<Subscriber> this_type;
                                        typedef rxu::decay_t<T> value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<T, this_type> observer_type;

                                        struct window_with_time_or_count_subscriber_values : public
window_with_time_or_count_values
```

```
                {
                    window_with_time_or_count_subscriber_values(composite_subscription cs,
dest_type d, window_with_time_or_count_values v, coordinator_type c)
                    : window_with_time_or_count_values(std::move(v))
                    , cs(std::move(cs))
                    , dest(std::move(d))
                    , coordinator(std::move(c))
                    , worker(coordinator.get_worker())
                    , cursor(0)
                    , subj_id(0)
                    {
                    }
                    composite_subscription cs;
                    dest_type dest;
                    coordinator_type coordinator;
                    rxsc::worker worker;
                    mutable int cursor;
                    mutable int subj_id;
                    mutable rxcpp::subjects::subject<T> subj;
                };
                typedef std::shared_ptr<window_with_time_or_count_subscriber_values> state_type;
                state_type state;

                window_with_time_or_count_observer(composite_subscription cs, dest_type d,
window_with_time_or_count_values v, coordinator_type c)
                    :
state(std::make_shared<window_with_time_or_count_subscriber_values>(window_with_time_or_count_subscriber_values(std::move(cs),
std::move(d), std::move(v), std::move(c))))
                {
                    auto new_id = state->subj_id;
                    auto produce_time = state->worker.now();
                    auto localState = state;

                    auto disposer = [=](const rxsc::schedulable&){
                        localState->cs.unsubscribe();
                        localState->dest.unsubscribe();
                        localState->worker.unsubscribe();
                    };
                    auto selectedDisposer = on_exception(
                        [&](){return localState->coordinator.act(disposer); },
                        localState->dest);
                    if (selectedDisposer.empty()) {
                        return;
                    }

                    localState->dest.add([=](){
                        localState->worker.schedule(selectedDisposer.get());
                    });
                    localState->cs.add([=](){
                        localState->worker.schedule(selectedDisposer.get());
                    });

                    //
                    // The scheduler is FIFO for any time T. Since the observer is scheduling
                    // on_next/on_error/oncompleted the timed schedule calls must be resheduled
                    // when they occur to ensure that production happens after
on_next/on_error/oncompleted
                    //

                    localState->worker.schedule(produce_time, [new_id, produce_time,
localState](const rxsc::schedulable&){
                        localState->worker.schedule(release_window(new_id, produce_time,
localState));
                    });
                }

                static std::function<void(const rxsc::schedulable&)> release_window(int id,
rxsc::scheduler::clock_type::time_point expected, state_type state) {
                    auto release = [id, expected, state](const rxsc::schedulable&) {
                        if (id != state->subj_id)
                            return;

                        state->subj.get_subscriber().on_completed();
                        state->subj = rxcpp::subjects::subject<T>();
                        state->dest.on_next(state->subj.get_observable().as_dynamic());
                        state->cursor = 0;
                        auto new_id = ++state->subj_id;
                        auto produce_time = expected + state->period;
                        state->worker.schedule(produce_time, [new_id, produce_time,
state](const rxsc::schedulable&){
                            state->worker.schedule(release_window(new_id,
produce_time, state));
```

```
                                                                        });
                                                                };
                                                                auto selectedRelease = on_exception(
                                                                        [&](){return state->coordinator.act(release); },
                                                                        state->dest);
                                                                if (selectedRelease.empty()) {
                                                                        return std::function<void(const rxsc::schedulable&)>();
                                                                }

                                                                return std::function<void(const rxsc::schedulable&)>(selectedRelease.get());
                                                        }

                                                        void on_next(T v) const {
                                                                auto localState = state;
                                                                auto work = [v, localState](const rxsc::schedulable& self){
                                                                        localState->subj.get_subscriber().on_next(v);
                                                                        if (++localState->cursor == localState->count) {
                                                                                release_window(localState->subj_id, localState->worker.now(), localState)(self);
                                                                        }
                                                                };
                                                                auto selectedWork = on_exception(
                                                                        [&](){return localState->coordinator.act(work); },
                                                                        localState->dest);
                                                                if (selectedWork.empty()) {
                                                                        return;
                                                                }
                                                                localState->worker.schedule(selectedWork.get());
                                                        }

                                                        void on_error(std::exception_ptr e) const {
                                                                auto localState = state;
                                                                auto work = [e, localState](const rxsc::schedulable&){
                                                                        localState->subj.get_subscriber().on_error(e);
                                                                        localState->dest.on_error(e);
                                                                };
                                                                auto selectedWork = on_exception(
                                                                        [&](){return localState->coordinator.act(work); },
                                                                        localState->dest);
                                                                if (selectedWork.empty()) {
                                                                        return;
                                                                }
                                                                localState->worker.schedule(selectedWork.get());
                                                        }

                                                        void on_completed() const {
                                                                auto localState = state;
                                                                auto work = [localState](const rxsc::schedulable&){
                                                                        localState->subj.get_subscriber().on_completed();
                                                                        localState->dest.on_completed();
                                                                };
                                                                auto selectedWork = on_exception(
                                                                        [&](){return localState->coordinator.act(work); },
                                                                        localState->dest);
                                                                if (selectedWork.empty()) {
                                                                        return;
                                                                }
                                                                localState->worker.schedule(selectedWork.get());
                                                        }

                                                        static subscriber<T, observer_type> make(dest_type d, window_with_time_or_count_values v) {
                                                                auto cs = composite_subscription();
                                                                auto coordinator = v.coordination.create_coordinator();

                                                                return make_subscriber<T>(cs, observer_type(this_type(cs, std::move(d), std::move(v), std::move(coordinator))));
                                                        }
                                                };

                                                template<class Subscriber>
                                                auto operator()(Subscriber dest) const
                                                        -> decltype(window_with_time_or_count_observer<Subscriber>::make(std::move(dest), initial)) {
                                                        return     window_with_time_or_count_observer<Subscriber>::make(std::move(dest), initial);
                                                }
                                        };

                                        template<class Duration, class Coordination>
                                        class window_with_time_or_count_factory
                                        {
```

```
                                        typedef rxu::decay_t<Duration> duration_type;
                                        typedef rxu::decay_t<Coordination> coordination_type;

                                        duration_type period;
                                        int count;
                                        coordination_type coordination;
                                public:
                                        window_with_time_or_count_factory(duration_type p, int n, coordination_type c) : period(p),
count(n), coordination(c) {}
                                        template<class Observable>
                                        auto operator()(Observable&& source)
                                                -> decltype(source.template
lift<observable<rxu::value_type_t<rxu::decay_t<Observable>>>>(window_with_time_or_count<rxu::value_type_t<rxu::decay_t<Observable>>,
Duration, Coordination>(period, count, coordination))) {
                                                return      source.template
lift<observable<rxu::value_type_t<rxu::decay_t<Observable>>>>(window_with_time_or_count<rxu::value_type_t<rxu::decay_t<Observable>>,
Duration, Coordination>(period, count, coordination));
                                        }
                                };

                        }

                        template<class Duration, class Coordination>
                        inline auto window_with_time_or_count(Duration period, int count, Coordination coordination)
                                ->      detail::window_with_time_or_count_factory<Duration, Coordination> {
                                return  detail::window_with_time_or_count_factory<Duration, Coordination>(period, count, coordination);
                        }

        }

}

#endif


#if !defined(RXCPP_OPERATORS_RX_WINDOW_TOGGLE_HPP)
#define RXCPP_OPERATORS_RX_WINDOW_TOGGLE_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class T, class Openings, class ClosingSelector, class Coordination>
                        struct window_toggle_traits {

                                using source_value_type = rxu::decay_t<T>;
                                using coordination_type = rxu::decay_t<Coordination>;
                                using openings_type = rxu::decay_t<Openings>;
                                using openings_value_type = typename openings_type::value_type;
                                using closing_selector_type = rxu::decay_t<ClosingSelector>;

                                static_assert(is_observable<openings_type>::value, "window_toggle Openings must be an
observable");

                                struct tag_not_valid {};
                                template<class CS, class CV>
                                static auto check(int) -> decltype((*(CS*)nullptr)((*(CV*)nullptr)));
                                template<class CS, class CV>
                                static tag_not_valid check(...);

                                static_assert(is_observable<decltype(check<closing_selector_type,
openings_value_type>(0))>::value, "window_toggle ClosingSelector must be a function with the signature
observable<U>(Openings::value_type)");

                                using closings_type = rxu::decay_t<decltype(check<closing_selector_type,
openings_value_type>(0))>;
                                using closings_value_type = typename closings_type::value_type;
                        };

                        template<class T, class Openings, class ClosingSelector, class Coordination>
                        struct window_toggle
                        {
                                typedef window_toggle<T, Openings, ClosingSelector, Coordination> this_type;

                                typedef window_toggle_traits<T, Openings, ClosingSelector, Coordination> traits;

                                using source_value_type = typename traits::source_value_type;
                                using coordination_type = typename traits::coordination_type;
```

```
                                        using coordinator_type = typename coordination_type::coordinator_type;
                                        using openings_type = typename traits::openings_type;
                                        using openings_value_type = typename traits::openings_value_type;
                                        using closing_selector_type = typename traits::closing_selector_type;
                                        using closings_value_type = typename traits::closings_value_type;

                                        struct window_toggle_values
                                        {
                                                window_toggle_values(openings_type opens, closing_selector_type closes,
coordination_type c)
                                                : openings(opens)
                                                , closingSelector(closes)
                                                , coordination(c)
                                                {
                                                }
                                                openings_type openings;
                                                mutable closing_selector_type closingSelector;
                                                coordination_type coordination;
                                        };
                                        window_toggle_values initial;

                                        window_toggle(openings_type opens, closing_selector_type closes, coordination_type coordination)
                                                : initial(opens, closes, coordination)
                                        {
                                        }

                                        template<class Subscriber>
                                        struct window_toggle_observer
                                        {
                                                typedef window_toggle_observer<Subscriber> this_type;
                                                typedef rxu::decay_t<T> value_type;
                                                typedef rxu::decay_t<Subscriber> dest_type;
                                                typedef observer<T, this_type> observer_type;

                                                struct window_toggle_subscriber_values : public window_toggle_values
                                                {
                                                        window_toggle_subscriber_values(composite_subscription cs, dest_type d,
window_toggle_values v, coordinator_type c)
                                                        : window_toggle_values(v)
                                                        , cs(std::move(cs))
                                                        , dest(std::move(d))
                                                        , coordinator(std::move(c))
                                                        , worker(coordinator.get_worker())
                                                        {
                                                        }
                                                        composite_subscription cs;
                                                        dest_type dest;
                                                        coordinator_type coordinator;
                                                        rxsc::worker worker;
                                                        mutable std::list<rxcpp::subjects::subject<T>> subj;
                                                };
                                                std::shared_ptr<window_toggle_subscriber_values> state;

                                                window_toggle_observer(composite_subscription cs, dest_type d, window_toggle_values
v, coordinator_type c)
                                                        :
state(std::make_shared<window_toggle_subscriber_values>(window_toggle_subscriber_values(std::move(cs), std::move(d), v, std::move(c))))
                                                {
                                                        auto localState = state;

                                                        composite_subscription innercs;

                                                        // when the out observer is unsubscribed all the
                                                        // inner subscriptions are unsubscribed as well
                                                        auto innerscope = localState->dest.add(innercs);

                                                        innercs.add([=](){
                                                                localState->dest.remove(innerscope);
                                                        });

                                                        auto source = on_exception(
                                                                [&](){return localState->coordinator.in(localState->openings); },
                                                                localState->dest);
                                                        if (source.empty()) {
                                                                return;
                                                        }

                                                        // this subscribe does not share the observer subscription
                                                        // so that when it is unsubscribed the observer can be called
                                                        // until the inner subscriptions have finished
                                                        auto sink = make_subscriber<openings_value_type>(
                                                                localState->dest,
```

```
                                                innercs,
                                                // on_next
                                                [localState](const openings_value_type& ov) {
                                                auto closer = localState->closingSelector(ov);

                                                auto it = localState->subj.insert(localState->subj.end(),
rxcpp::subjects::subject<T>());
                                                localState->dest.on_next(it->get_observable().as_dynamic());

                                                composite_subscription innercs;

                                                // when the out observer is unsubscribed all the
                                                // inner subscriptions are unsubscribed as well
                                                auto innerscope = localState->dest.add(innercs);

                                                innercs.add([=](){
                                                        localState->dest.remove(innerscope);
                                                });

                                                auto source = localState->coordinator.in(closer);

                                                auto sit = std::make_shared<decltype(it)>(it);
                                                auto close = [localState, sit]() {
                                                        auto it = *sit;
                                                        *sit = localState->subj.end();
                                                        if (it != localState->subj.end()) {
                                                                it->get_subscriber().on_completed();
                                                                localState->subj.erase(it);
                                                        }
                                                };

                                                // this subscribe does not share the observer subscription
                                                // so that when it is unsubscribed the observer can be called
                                                // until the inner subscriptions have finished
                                                auto sink = make_subscriber<closings_value_type>(
                                                        localState->dest,
                                                        innercs,
                                                        // on_next
                                                        [close, innercs](closings_value_type) {
                                                        close();
                                                        innercs.unsubscribe();
                                                },
                                                        // on_error
                                                        [localState](std::exception_ptr e) {
                                                        localState->dest.on_error(e);
                                                },
                                                        // on_completed
                                                        close
                                                        );
                                                auto selectedSink = localState->coordinator.out(sink);
                                                source.subscribe(std::move(selectedSink));
                                        },
                                                // on_error
                                                [localState](std::exception_ptr e) {
                                                localState->dest.on_error(e);
                                        },
                                                // on_completed
                                                []() {
                                        }
                                        );
                                        auto selectedSink = on_exception(
                                                [&](){return localState->coordinator.out(sink); },
                                                localState->dest);
                                        if (selectedSink.empty()) {
                                                return;
                                        }
                                        source->subscribe(std::move(selectedSink.get()));
                                }

                                void on_next(T v) const {
                                        auto localState = state;
                                        auto work = [v, localState](const rxsc::schedulable&){
                                                for (auto s : localState->subj) {
                                                        s.get_subscriber().on_next(v);
                                                }
                                        };
                                        auto selectedWork = on_exception(
                                                [&](){return localState->coordinator.act(work); },
                                                localState->dest);
                                        if (selectedWork.empty()) {
                                                return;
                                        }
```

```
                                                localState->worker.schedule(selectedWork.get());
                                        }

                                        void on_error(std::exception_ptr e) const {
                                                auto localState = state;
                                                auto work = [e, localState](const rxsc::schedulable&){
                                                        for (auto s : localState->subj) {
                                                                s.get_subscriber().on_error(e);
                                                        }
                                                        localState->dest.on_error(e);
                                                };
                                                auto selectedWork = on_exception(
                                                        [&](){return localState->coordinator.act(work); },
                                                        localState->dest);
                                                if (selectedWork.empty()) {
                                                        return;
                                                }
                                                localState->worker.schedule(selectedWork.get());
                                        }

                                        void on_completed() const {
                                                auto localState = state;
                                                auto work = [localState](const rxsc::schedulable&){
                                                        for (auto s : localState->subj) {
                                                                s.get_subscriber().on_completed();
                                                        }
                                                        localState->dest.on_completed();
                                                };
                                                auto selectedWork = on_exception(
                                                        [&](){return localState->coordinator.act(work); },
                                                        localState->dest);
                                                if (selectedWork.empty()) {
                                                        return;
                                                }
                                                localState->worker.schedule(selectedWork.get());
                                        }

                                        static subscriber<T, observer_type> make(dest_type d, window_toggle_values v) {
                                                auto cs = composite_subscription();
                                                auto coordinator = v.coordination.create_coordinator();

                                                return make_subscriber<T>(cs, observer_type(this_type(cs, std::move(d),
std::move(v), std::move(coordinator))));
                                        }
                                };

                                template<class Subscriber>
                                auto operator()(Subscriber dest) const
                                        -> decltype(window_toggle_observer<Subscriber>::make(std::move(dest), initial)) {
                                        return  window_toggle_observer<Subscriber>::make(std::move(dest), initial);
                                }
                        };

                        template<class Openings, class ClosingSelector, class Coordination>
                        class window_toggle_factory
                        {
                                typedef rxu::decay_t<Openings> openings_type;
                                typedef rxu::decay_t<ClosingSelector> closing_selector_type;
                                typedef rxu::decay_t<Coordination> coordination_type;

                                openings_type openings;
                                closing_selector_type closingSelector;
                                coordination_type coordination;
                        public:
                                window_toggle_factory(openings_type opens, closing_selector_type closes, coordination_type c) :
openings(opens), closingSelector(closes), coordination(c) {}
                                template<class Observable>
                                auto operator()(Observable&& source)
                                        -> decltype(source.template
lift<observable<rxu::value_type_t<rxu::decay_t<Observable>>>>(window_toggle<rxu::value_type_t<rxu::decay_t<Observable>>, Openings,
ClosingSelector, Coordination>(openings, closingSelector, coordination))) {
                                        return  source.template
lift<observable<rxu::value_type_t<rxu::decay_t<Observable>>>>(window_toggle<rxu::value_type_t<rxu::decay_t<Observable>>, Openings,
ClosingSelector, Coordination>(openings, closingSelector, coordination));
                                }
                        };

                }

                template<class Openings, class ClosingSelector, class Coordination>
                inline auto window_toggle(Openings openings, ClosingSelector closingSelector, Coordination coordination)
                        ->      detail::window_toggle_factory<Openings, ClosingSelector, Coordination> {
```

```
                return  detail::window_toggle_factory<Openings, ClosingSelector, Coordination>(openings, closingSelector,
coordination);
            }

            template<class Openings, class ClosingSelector>
            inline auto window_toggle(Openings openings, ClosingSelector closingSelector)
                ->      detail::window_toggle_factory<Openings, ClosingSelector, identity_one_worker> {
                return  detail::window_toggle_factory<Openings, ClosingSelector, identity_one_worker>(openings,
closingSelector, identity_immediate());
            }

        }

}

#endif


namespace rxcpp {

    struct all_tag {
        template<class Included>
        struct include_header{
            static_assert(Included::value, "missing include: please #include <rxcpp/operators/rx-all.hpp>");
        };
    };

    struct any_tag {
        template<class Included>
        struct include_header{
            static_assert(Included::value, "missing include: please #include <rxcpp/operators/rx-any.hpp>");
        };
    };

    struct exists_tag : any_tag {};
    struct contains_tag : any_tag {};

    struct combine_latest_tag {
        template<class Included>
        struct include_header{
            static_assert(Included::value, "missing include: please #include <rxcpp/operators/rx-combine_latest.hpp>");
        };
    };

    struct debounce_tag {
        template<class Included>
        struct include_header{
            static_assert(Included::value, "missing include: please #include <rxcpp/operators/rx-debounce.hpp>");
        };
    };

    struct delay_tag {
        template<class Included>
        struct include_header{
            static_assert(Included::value, "missing include: please #include <rxcpp/operators/rx-delay.hpp>");
        };
    };

    struct distinct_tag {
        template<class Included>
        struct include_header{
            static_assert(Included::value, "missing include: please #include <rxcpp/operators/rx-distinct.hpp>");
        };
    };

    struct distinct_until_changed_tag {
        template<class Included>
        struct include_header{
            static_assert(Included::value, "missing include: please #include <rxcpp/operators/rx-
distinct_until_changed.hpp>");
        };
    };

    struct element_at_tag {
        template<class Included>
        struct include_header{
            static_assert(Included::value, "missing include: please #include <rxcpp/operators/rx-element_at.hpp>");
        };
    };

    struct filter_tag {
        template<class Included>
```

```
                struct include_header{
                        static_assert(Included::value, "missing include: please #include <rxcpp/operators/rx-filter.hpp>");
                };
        };

        struct group_by_tag {
                template<class Included>
                struct include_header{
                        static_assert(Included::value, "missing include: please #include <rxcpp/operators/rx-group_by.hpp>");
                };
        };

        struct ignore_elements_tag {
                template<class Included>
                struct include_header{
                        static_assert(Included::value, "missing include: please #include <rxcpp/operators/rx-ignore_elements.hpp>");
                };
        };

        class empty_error : public std::runtime_error
        {
        public:
                explicit empty_error(const std::string& msg) :
                        std::runtime_error(msg)
                {}
        };
        struct reduce_tag {
                template<class Included>
                struct include_header{
                        static_assert(Included::value, "missing include: please #include <rxcpp/operators/rx-reduce.hpp>");
                };
        };
        struct first_tag  {};
        struct last_tag : reduce_tag {};
        struct sum_tag : reduce_tag {};
        struct average_tag : reduce_tag {};
        struct min_tag : reduce_tag {};
        struct max_tag : reduce_tag {};

        struct with_latest_from_tag {
                template<class Included>
                struct include_header{
                        static_assert(Included::value, "missing include: please #include <rxcpp/operators/rx-with_latest_from.hpp>");
                };
        };

        struct zip_tag {
                template<class Included>
                struct include_header{
                        static_assert(Included::value, "missing include: please #include <rxcpp/operators/rx-zip.hpp>");
                };
        };

}

#endif


#if !defined(RXCPP_RX_OBSERVABLE_HPP)
#define RXCPP_RX_OBSERVABLE_HPP

//_include "rx-includes.hpp"

#ifdef __GNUG__
#define EXPLICIT_THIS this->
#else
#define EXPLICIT_THIS
#endif

namespace rxcpp {

        namespace detail {

                template<class Subscriber, class T>
                struct has_on_subscribe_for
                {
                        struct not_void {};
                        template<class CS, class CT>
                        static auto check(int) -> decltype((*(CT*)nullptr).on_subscribe(*(CS*)nullptr));
                        template<class CS, class CT>
                        static not_void check(...);
```

```
                                typedef decltype(check<rxu::decay_t<Subscriber>, T>(0)) detail_result;
                                static const bool value = std::is_same<detail_result, void>::value;
                        };

        }

        template<class T>
        class dynamic_observable
                : public rxs::source_base<T>
        {
                struct state_type
                : public std::enable_shared_from_this<state_type>
                {
                        typedef std::function<void(subscriber<T>)> onsubscribe_type;

                        onsubscribe_type on_subscribe;
                };
                std::shared_ptr<state_type> state;

                template<class U>
                friend bool operator==(const dynamic_observable<U>&, const dynamic_observable<U>&);

                template<class SO>
                void construct(SO&& source, rxs::tag_source&&) {
                        rxu::decay_t<SO> so = std::forward<SO>(source);
                        state->on_subscribe = [so](subscriber<T> o) mutable {
                                so.on_subscribe(std::move(o));
                        };
                }

                struct tag_function {};
                template<class F>
                void construct(F&& f, tag_function&&) {
                        state->on_subscribe = std::forward<F>(f);
                }

        public:

                typedef tag_dynamic_observable dynamic_observable_tag;

                dynamic_observable()
                {
                }

                template<class SOF>
                explicit dynamic_observable(SOF&& sof, typename std::enable_if<!is_dynamic_observable<SOF>::value, void**>::type =
0)
                        : state(std::make_shared<state_type>())
                {
                        construct(std::forward<SOF>(sof),
                                typename std::conditional<rxs::is_source<SOF>::value || rxo::is_operator<SOF>::value,
rxs::tag_source, tag_function>::type());
                }

                void on_subscribe(subscriber<T> o) const {
                        state->on_subscribe(std::move(o));
                }

                template<class Subscriber>
                typename std::enable_if<is_subscriber<Subscriber>::value, void>::type
                        on_subscribe(Subscriber o) const {
                                state->on_subscribe(o.as_dynamic());
                        }
        };

        template<class T>
        inline bool operator==(const dynamic_observable<T>& lhs, const dynamic_observable<T>& rhs) {
                return lhs.state == rhs.state;
        }
        template<class T>
        inline bool operator!=(const dynamic_observable<T>& lhs, const dynamic_observable<T>& rhs) {
                return !(lhs == rhs);
        }

        template<class T, class Source>
        observable<T> make_observable_dynamic(Source&& s) {
                return observable<T>(dynamic_observable<T>(std::forward<Source>(s)));
        }

        namespace detail {
                template<bool Selector, class Default, class SO>
                struct resolve_observable;
```

```
		template<class Default, class SO>
		struct resolve_observable<true, Default, SO>
		{
			typedef typename SO::type type;
			typedef typename type::value_type value_type;
			static const bool value = true;
			typedef observable<value_type, type> observable_type;
			template<class... AN>
			static observable_type make(const Default&, AN&&... an) {
				return observable_type(type(std::forward<AN>(an)...));
			}
		};
		template<class Default, class SO>
		struct resolve_observable<false, Default, SO>
		{
			static const bool value = false;
			typedef Default observable_type;
			template<class... AN>
			static observable_type make(const observable_type& that, const AN&...) {
				return that;
			}
		};
		template<class SO>
		struct resolve_observable<true, void, SO>
		{
			typedef typename SO::type type;
			typedef typename type::value_type value_type;
			static const bool value = true;
			typedef observable<value_type, type> observable_type;
			template<class... AN>
			static observable_type make(AN&&... an) {
				return observable_type(type(std::forward<AN>(an)...));
			}
		};
		template<class SO>
		struct resolve_observable<false, void, SO>
		{
			static const bool value = false;
			typedef void observable_type;
			template<class... AN>
			static observable_type make(const AN&...) {
			}
		};

	}

	template<class Selector, class Default, template<class... TN> class SO, class... AN>
	struct defer_observable
		: public detail::resolve_observable<Selector::value, Default, rxu::defer_type<SO, AN...>>
	{
	};

	/*!
	\brief a source of values whose methods block until all values have been emitted. subscribe or use one of the operator methods that reduce the values emitted to a single value.

	\ingroup group-observable

	*/
	template<class T, class Observable>
	class blocking_observable
	{
		template<class Obsvbl, class... ArgN>
		static auto blocking_subscribe(const Obsvbl& source, bool do_rethrow, ArgN&&... an)
			-> void {
			std::mutex lock;
			std::condition_variable wake;
			std::exception_ptr error;

			struct tracking
			{
				~tracking()
				{
					if (!disposed || !wakened) std::terminate();
				}
				tracking()
				{
					disposed = false;
					wakened = false;
					false_wakes = 0;
					true_wakes = 0;
```

```
                        }
                        std::atomic_bool disposed;
                        std::atomic_bool wakened;
                        std::atomic_int false_wakes;
                        std::atomic_int true_wakes;
                };
                auto track = std::make_shared<tracking>();

                auto dest = make_subscriber<T>(std::forward<ArgN>(an)...);

                // keep any error to rethrow at the end.
                auto scbr = make_subscriber<T>(
                        dest,
                        [&](T t){dest.on_next(t); },
                        [&](std::exception_ptr e){
                        if (do_rethrow) {
                                error = e;
                        }
                        else {
                                dest.on_error(e);
                        }
                },
                        [&](){dest.on_completed(); }
                );

                auto cs = scbr.get_subscription();
                cs.add(
                        [&, track](){
                        // OSX geting invalid x86 op if notify_one is after the disposed = true
                        // presumably because the condition_variable may already have been awakened
                        // and is now sitting in a while loop on disposed
                        wake.notify_one();
                        track->disposed = true;
                });

                std::unique_lock<std::mutex> guard(lock);
                source.subscribe(std::move(scbr));

                wake.wait(guard,
                        [&, track](){
                        // this is really not good.
                        // false wakeups were never followed by true wakeups so..

                        // anyways this gets triggered before disposed is set now so wait.
                        while (!track->disposed) {
                                ++track->false_wakes;
                        }
                        ++track->true_wakes;
                        return true;
                });
                track->wakened = true;
                if (!track->disposed || !track->wakened) std::terminate();

                if (error) { std::rethrow_exception(error); }
        }

public:
        typedef rxu::decay_t<Observable> observable_type;
        observable_type source;
        ~blocking_observable()
        {
        }
        blocking_observable(observable_type s) : source(std::move(s)) {}

        ///
        /// `subscribe` will cause this observable to emit values to the provided subscriber.
        ///
        /// \return void
        ///
        /// \param an... - the arguments are passed to make_subscriber().
        ///
        /// callers must provide enough arguments to make a subscriber.
        /// overrides are supported. thus
        ///   `subscribe(thesubscriber, composite_subscription())`
        /// will take `thesubscriber.get_observer()` and the provided
        /// subscription and subscribe to the new subscriber.
        /// the `on_next`, `on_error`, `on_completed` methods can be supplied instead of an observer
        /// if a subscription or subscriber is not provided then a new subscription will be created.
        ///
        template<class... ArgN>
        auto subscribe(ArgN&&... an) const
                -> void {
```

```
        return blocking_subscribe(source, false, std::forward<ArgN>(an)...);
}

///
/// `subscribe_with_rethrow` will cause this observable to emit values to the provided subscriber.
///
/// \note  If the source observable calls on_error, the raised exception is rethrown by this method.
///
/// \note  If the source observable calls on_error, the `on_error` method on the subscriber will not be called.
///
/// \return void
///
/// \param an... - the arguments are passed to make_subscriber().
///
/// callers must provide enough arguments to make a subscriber.
/// overrides are supported. thus
///   `subscribe(thesubscriber, composite_subscription())`
/// will take `thesubscriber.get_observer()` and the provided
/// subscription and subscribe to the new subscriber.
/// the `on_next`, `on_error`, `on_completed` methods can be supplied instead of an observer
/// if a subscription or subscriber is not provided then a new subscription will be created.
///
template<class... ArgN>
auto subscribe_with_rethrow(ArgN&&... an) const
        -> void {
        return blocking_subscribe(source, true, std::forward<ArgN>(an)...);
}

/*! Return the first item emitted by this blocking_observable, or throw an std::runtime_error exception if it emits no items.

\return  The first item emitted by this blocking_observable.

\note  If the source observable calls on_error, the raised exception is rethrown by this method.

\sample
When the source observable emits at least one item:
\snippet blocking_observable.cpp blocking first sample
\snippet output.txt blocking first sample

When the source observable is empty:
\snippet blocking_observable.cpp blocking first empty sample
\snippet output.txt blocking first empty sample
*/
template<class... AN>
auto first(AN**...) -> delayed_type_t<T, AN...> const {
        rxu::maybe<T> result;
        composite_subscription cs;
        subscribe_with_rethrow(
                cs,
                [&](T v){result.reset(v); cs.unsubscribe(); });
        if (result.empty())
                throw rxcpp::empty_error("first() requires a stream with at least one value");
        return result.get();
        static_assert(sizeof...(AN) == 0, "first() was passed too many arguments.");
}

/*! Return the last item emitted by this blocking_observable, or throw an std::runtime_error exception if it emits no items.

\return  The last item emitted by this blocking_observable.

\note  If the source observable calls on_error, the raised exception is rethrown by this method.

\sample
When the source observable emits at least one item:
\snippet blocking_observable.cpp blocking last sample
\snippet output.txt blocking last sample

When the source observable is empty:
\snippet blocking_observable.cpp blocking last empty sample
\snippet output.txt blocking last empty sample
*/
template<class... AN>
auto last(AN**...) -> delayed_type_t<T, AN...> const {
        rxu::maybe<T> result;
        subscribe_with_rethrow(
                [&](T v){result.reset(v); });
        if (result.empty())
                throw rxcpp::empty_error("last() requires a stream with at least one value");
        return result.get();
        static_assert(sizeof...(AN) == 0, "last() was passed too many arguments.");
}
```

```
		/*! Return the total number of items emitted by this blocking_observable.

		\return  The total number of items emitted by this blocking_observable.

		\sample
		\snippet blocking_observable.cpp blocking count sample
		\snippet output.txt blocking count sample

		When the source observable calls on_error:
		\snippet blocking_observable.cpp blocking count error sample
		\snippet output.txt blocking count error sample
		*/
		int count() const {
			int result = 0;
			source.count().as_blocking().subscribe_with_rethrow(
				[&](int v){result = v; });
			return result;
		}

		/*! Return the sum of all items emitted by this blocking_observable, or throw an std::runtime_error exception if it emits no items.

		\return  The sum of all items emitted by this blocking_observable.

		\sample
		When the source observable emits at least one item:
		\snippet blocking_observable.cpp blocking sum sample
		\snippet output.txt blocking sum sample

		When the source observable is empty:
		\snippet blocking_observable.cpp blocking sum empty sample
		\snippet output.txt blocking sum empty sample

		When the source observable calls on_error:
		\snippet blocking_observable.cpp blocking sum error sample
		\snippet output.txt blocking sum error sample
		*/
		T sum() const {
			return source.sum().as_blocking().last();
		}

		/*! Return the average value of all items emitted by this blocking_observable, or throw an std::runtime_error exception if it emits no items.

		\return  The average value of all items emitted by this blocking_observable.

		\sample
		When the source observable emits at least one item:
		\snippet blocking_observable.cpp blocking average sample
		\snippet output.txt blocking average sample

		When the source observable is empty:
		\snippet blocking_observable.cpp blocking average empty sample
		\snippet output.txt blocking average empty sample

		When the source observable calls on_error:
		\snippet blocking_observable.cpp blocking average error sample
		\snippet output.txt blocking average error sample
		*/
		double average() const {
			return source.average().as_blocking().last();
		}

		/*! Return the max of all items emitted by this blocking_observable, or throw an std::runtime_error exception if it emits no items.

		\return  The max of all items emitted by this blocking_observable.

		\sample
		When the source observable emits at least one item:
		\snippet blocking_observable.cpp blocking max sample
		\snippet output.txt blocking max sample

		When the source observable is empty:
		\snippet blocking_observable.cpp blocking max empty sample
		\snippet output.txt blocking max empty sample

		When the source observable calls on_error:
		\snippet blocking_observable.cpp blocking max error sample
		\snippet output.txt blocking max error sample
		*/
		T max() const {
```

```
                        return source.max().as_blocking().last();
                }

                /*! Return the min of all items emitted by this blocking_observable, or throw an std::runtime_error exception if it emits no items.

                \return  The min of all items emitted by this blocking_observable.

                \sample
                When the source observable emits at least one item:
                \snippet blocking_observable.cpp blocking min sample
                \snippet output.txt blocking min sample

                When the source observable is empty:
                \snippet blocking_observable.cpp blocking min empty sample
                \snippet output.txt blocking min empty sample

                When the source observable calls on_error:
                \snippet blocking_observable.cpp blocking min error sample
                \snippet output.txt blocking min error sample
                */
                T min() const {
                        return source.min().as_blocking().last();
                }
        };

        namespace detail {

                template<class SourceOperator, class Subscriber>
                struct safe_subscriber
                {
                        safe_subscriber(SourceOperator& so, Subscriber& o) : so(std::addressof(so)), o(std::addressof(o)) {}

                        void subscribe() {
                                try {
                                        so->on_subscribe(*o);
                                }
                                catch (...) {
                                        if (!o->is_subscribed()) {
                                                throw;
                                        }
                                        o->on_error(std::current_exception());
                                        o->unsubscribe();
                                }
                        }

                        void operator()(const rxsc::schedulable&) {
                                subscribe();
                        }

                        SourceOperator* so;
                        Subscriber* o;
                };

        }

        template<>
        class observable<void, void>;

        /*!
        \defgroup group-observable Observables

        \brief These are the set of observable classes in rxcpp.

        \class rxcpp::observable

        \ingroup group-observable group-core

        \brief a source of values. subscribe or use one of the operator methods that return a new observable, which uses this observable as a source.

        \par Some code
        This sample will observable::subscribe() to values from a observable<void, void>::range().

        \sample
        \snippet range.cpp range sample
        \snippet output.txt range sample

        */
        template<class T, class SourceOperator>
        class observable
                : public observable_base<T>
```

```
	{
		static_assert(std::is_same<T, typename SourceOperator::value_type>::value, "SourceOperator::value_type must be the same as T in observable<T, SourceOperator>");

		typedef observable<T, SourceOperator> this_type;

	public:
		typedef rxu::decay_t<SourceOperator> source_operator_type;
		mutable source_operator_type source_operator;

	private:

		template<class U, class SO>
		friend class observable;

		template<class U, class SO>
		friend bool operator==(const observable<U, SO>&, const observable<U, SO>&);

		template<class Subscriber>
		auto detail_subscribe(Subscriber o) const
			-> composite_subscription {

			typedef rxu::decay_t<Subscriber> subscriber_type;

			static_assert(is_subscriber<subscriber_type>::value, "subscribe must be passed a subscriber");
			static_assert(std::is_same<typename source_operator_type::value_type, T>::value && std::is_convertible<T*, typename subscriber_type::value_type*>::value, "the value types in the sequence must match or be convertible");
			static_assert(detail::has_on_subscribe_for<subscriber_type, source_operator_type>::value, "inner must have on_subscribe method that accepts this subscriber ");

			trace_activity().subscribe_enter(*this, o);

			if (!o.is_subscribed()) {
				trace_activity().subscribe_return(*this);
				return o.get_subscription();
			}

			detail::safe_subscriber<source_operator_type, subscriber_type> subscriber(source_operator, o);

			// make sure to let current_thread take ownership of the thread as early as possible.
			if (rxsc::current_thread::is_schedule_required()) {
				const auto& sc = rxsc::make_current_thread();
				sc.create_worker(o.get_subscription()).schedule(subscriber);
			}
			else {
				// current_thread already owns this thread.
				subscriber.subscribe();
			}

			trace_activity().subscribe_return(*this);
			return o.get_subscription();
		}

	public:
		typedef T value_type;

		static_assert(rxo::is_operator<source_operator_type>::value || rxs::is_source<source_operator_type>::value, "observable must wrap an operator or source");

		~observable()
		{
		}

		observable()
		{
		}

		explicit observable(const source_operator_type& o)
			: source_operator(o)
		{
		}
		explicit observable(source_operator_type&& o)
			: source_operator(std::move(o))
		{
		}

		/// implicit conversion between observables of the same value_type
		template<class SO>
		observable(const observable<T, SO>& o)
			: source_operator(o.source_operator)
		{}
		/// implicit conversion between observables of the same value_type
```

```
                        template<class SO>
                        observable(observable<T, SO>&& o)
                                : source_operator(std::move(o.source_operator))
                        {}

#if 0
                        template<class I>
                        void on_subscribe(observer<T, I> o) const {
                                source_operator.on_subscribe(o);
                        }
#endif

                        /*! Return a new observable that performs type-forgetting conversion of this observable.

                        \return  The source observable converted to observable<T>.

                        \note This operator could be useful to workaround lambda deduction bug on msvc 2013.

                        \sample
                        \snippet as_dynamic.cpp as_dynamic sample
                        \snippet output.txt as_dynamic sample
                        */
                        template<class... AN>
                        observable<T> as_dynamic(AN**...) const {
                                return *this;
                                static_assert(sizeof...(AN) == 0, "as_dynamic() was passed too many arguments.");
                        }

                        /*! Return a new observable that contains the blocking methods for this observable.

                        \return  An observable that contains the blocking methods for this observable.

                        \sample
                        \snippet from.cpp threaded from sample
                        \snippet output.txt threaded from sample
                        */
                        template<class... AN>
                        blocking_observable<T, this_type> as_blocking(AN**...) const {
                                return blocking_observable<T, this_type>(*this);
                                static_assert(sizeof...(AN) == 0, "as_blocking() was passed too many arguments.");
                        }

                        /// \cond SHOW_SERVICE_MEMBERS

                        ///
                        /// takes any function that will take this observable and produce a result value.
                        /// this is intended to allow externally defined operators, that use subscribe,
                        /// to be connected into the expression.
                        ///
                        template<class OperatorFactory>
                        auto op(OperatorFactory&& of) const
                                -> decltype(of(*(const this_type*)nullptr)) {
                                return      of(*this);
                                static_assert(is_operator_factory_for<this_type, OperatorFactory>::value, "Function passed for op() must have
the signature Result(SourceObservable)");
                        }

                        ///
                        /// takes any function that will take a subscriber for this observable and produce a subscriber.
                        /// this is intended to allow externally defined operators, that use make_subscriber, to be connected
                        /// into the expression.
                        ///
                        template<class ResultType, class Operator>
                        auto lift(Operator&& op) const
                                ->      observable<rxu::value_type_t<rxo::detail::lift_operator<ResultType, source_operator_type, Operator>>,
rxo::detail::lift_operator<ResultType, source_operator_type, Operator>> {
                                return  observable<rxu::value_type_t<rxo::detail::lift_operator<ResultType, source_operator_type, Operator>>,
rxo::detail::lift_operator<ResultType, source_operator_type, Operator>>(
                                        rxo::detail::lift_operator<ResultType, source_operator_type, Operator>(source_operator,
std::forward<Operator>(op)));
                                static_assert(detail::is_lift_function_for<T, subscriber<ResultType>, Operator>::value, "Function passed for
lift() must have the signature subscriber<...>(subscriber<T, ...>)");
                        }

                        ///
                        /// takes any function that will take a subscriber for this observable and produce a subscriber.
                        /// this is intended to allow externally defined operators, that use make_subscriber, to be connected
                        /// into the expression.
                        ///
                        template<class ResultType, class Operator>
                        auto lift_if(Operator&& op) const
                                -> typename std::enable_if<detail::is_lift_function_for<T, subscriber<ResultType>, Operator>::value,
```

```
                observable<rxu::value_type_t<rxo::detail::lift_operator<ResultType, source_operator_type, Operator>>,
rxo::detail::lift_operator<ResultType, source_operator_type, Operator>>>::type {
                return  observable<rxu::value_type_t<rxo::detail::lift_operator<ResultType, source_operator_type, Operator>>,
rxo::detail::lift_operator<ResultType, source_operator_type, Operator>>(
                    rxo::detail::lift_operator<ResultType, source_operator_type, Operator>(source_operator,
std::forward<Operator>(op)));
            }
            ///
            /// takes any function that will take a subscriber for this observable and produce a subscriber.
            /// this is intended to allow externally defined operators, that use make_subscriber, to be connected
            /// into the expression.
            ///
            template<class ResultType, class Operator>
            auto lift_if(Operator&&) const
                -> typename std::enable_if<!detail::is_lift_function_for<T, subscriber<ResultType>, Operator>::value,
                decltype(rxs::from<ResultType>())>::type {
                return      rxs::from<ResultType>();
            }
            /// \endcond

            /*! Subscribe will cause this observable to emit values to the provided subscriber.

            \tparam ArgN  types of the subscriber parameters

            \param an  the parameters for making a subscriber

            \return  A subscription with which the observer can stop receiving items before the observable has finished sending them.

            The arguments of subscribe are forwarded to rxcpp::make_subscriber function. Some possible alternatives are:

            - Pass an already composed rxcpp::subscriber:
            \snippet subscribe.cpp subscribe by subscriber
            \snippet output.txt subscribe by subscriber

            - Pass an rxcpp::observer. This allows subscribing the same subscriber to several observables:
            \snippet subscribe.cpp subscribe by observer
            \snippet output.txt subscribe by observer

            - Pass an `on_next` handler:
            \snippet subscribe.cpp subscribe by on_next
            \snippet output.txt subscribe by on_next

            - Pass `on_next` and `on_error` handlers:
            \snippet subscribe.cpp subscribe by on_next and on_error
            \snippet output.txt subscribe by on_next and on_error

            - Pass `on_next` and `on_completed` handlers:
            \snippet subscribe.cpp subscribe by on_next and on_completed
            \snippet output.txt subscribe by on_next and on_completed

            - Pass `on_next`, `on_error`, and `on_completed` handlers:
            \snippet subscribe.cpp subscribe by on_next, on_error, and on_completed
            \snippet output.txt subscribe by on_next, on_error, and on_completed
            .

            All the alternatives above also support passing rxcpp::composite_subscription instance. For example:
            \snippet subscribe.cpp subscribe by subscription, on_next, and on_completed
            \snippet output.txt subscribe by subscription, on_next, and on_completed

            If neither subscription nor subscriber are provided, then a new subscription is created and returned as a result:
            \snippet subscribe.cpp subscribe unsubscribe
            \snippet output.txt subscribe unsubscribe

            For more details, see rxcpp::make_subscriber function description.
            */
            template<class... ArgN>
            auto subscribe(ArgN&&... an) const
                -> composite_subscription {
                return detail_subscribe(make_subscriber<T>(std::forward<ArgN>(an)...));
            }

            /*! @copydoc rx-all.hpp
            */
            template<class... AN>
            auto all(AN&&... an) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(observable_member(all_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
                /// \endcond
            {
                return  observable_member(all_tag{}, *this, std::forward<AN>(an)...);
            }
```

```
                        /*! @copydoc rxcpp::operators::exists
                        */
                        template<class... AN>
                        auto exists(AN&&... an) const
                                /// \cond SHOW_SERVICE_MEMBERS
                                -> decltype(observable_member(exists_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
                                /// \endcond
                        {
                                return  observable_member(exists_tag{}, *this, std::forward<AN>(an)...);
                        }

                        /*! @copydoc rxcpp::operators::contains
                        */
                        template<class... AN>
                        auto contains(AN&&... an) const
                                /// \cond SHOW_SERVICE_MEMBERS
                                -> decltype(observable_member(contains_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
                                /// \endcond
                        {
                                return  observable_member(contains_tag{}, *this, std::forward<AN>(an)...);
                        }

                        /*! @copydoc rx-filter.hpp
                        */
                        template<class... AN>
                        auto filter(AN&&... an) const
                                /// \cond SHOW_SERVICE_MEMBERS
                                -> decltype(observable_member(filter_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
                                /// \endcond
                        {
                                return  observable_member(filter_tag{}, *this, std::forward<AN>(an)...);
                        }

                        /*! If the source Observable terminates without emitting any items, emits items from a backup Observable.

                        \tparam BackupSource  the type of the backup observable.

                        \param t  a backup observable that is used if the source observable is empty.

                        \return  Observable that emits items from a backup observable if the source observable is empty.

                        \sample
                        \snippet switch_if_empty.cpp switch_if_empty sample
                        \snippet output.txt switch_if_empty sample
                        */
                        template<class BackupSource>
                        auto switch_if_empty(BackupSource t) const
                                /// \cond SHOW_SERVICE_MEMBERS
                                -> typename std::enable_if<is_observable<BackupSource>::value,
                                decltype(EXPLICIT_THIS lift<T>(rxo::detail::switch_if_empty<T, BackupSource>(std::move(t))))>::type
                                /// \endcond
                        {
                                return                  lift<T>(rxo::detail::switch_if_empty<T, BackupSource>(std::move(t)));
                        }

                        /*! If the source Observable terminates without emitting any items, emits a default item and completes.

                        \tparam V  the type of the value to emit.

                        \param v  the default value to emit

                        \return  Observable that emits the specified default item if the source observable is empty.

                        \sample
                        \snippet default_if_empty.cpp default_if_empty sample
                        \snippet output.txt default_if_empty sample
                        */
                        template <typename V>
                        auto default_if_empty(V v) const
                                -> decltype(EXPLICIT_THIS switch_if_empty(rxs::from(std::move(v))))
                        {
                                return                  switch_if_empty(rxs::from(std::move(v)));
                        }

                        /*! Determine whether two Observables emit the same sequence of items.

                        \tparam OtherSource      the type of the other observable.
                        \tparam BinaryPredicate  the type of the value comparing function. The signature should be equivalent to the following: bool
pred(const T1& a, const T2& b);
                        \tparam Coordination  the type of the scheduler.

                        \param t     the other Observable that emits items to compare.
```

```
                \param pred  the function that implements comparison of two values.
                \param cn    the scheduler.

                \return  Observable that emits true only if both sequences terminate normally after emitting the same sequence of items in the same order; otherwise it will emit false.

                \sample
                \snippet sequence_equal.cpp sequence_equal sample
                \snippet output.txt sequence_equal sample
                */
                template<class OtherSource, class BinaryPredicate, class Coordination>
                auto sequence_equal(OtherSource&& t, BinaryPredicate&& pred, Coordination&& cn) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> typename std::enable_if<is_observable<OtherSource>::value,
                        observable<bool, rxo::detail::sequence_equal<T, this_type, OtherSource, BinaryPredicate, Coordination>>>::type
                        /// \endcond
                {
                        return  observable<bool, rxo::detail::sequence_equal<T, this_type, OtherSource, BinaryPredicate, Coordination>>(
                                        rxo::detail::sequence_equal<T, this_type, OtherSource, BinaryPredicate, Coordination>(*this, std::forward<OtherSource>(t), std::forward<BinaryPredicate>(pred), std::forward<Coordination>(cn)));
                }


                /*! Determine whether two Observables emit the same sequence of items.

                \tparam OtherSource      the type of the other observable.
                \tparam BinaryPredicate  the type of the value comparing function. The signature should be equivalent to the following: bool pred(const T1& a, const T2& b);

                \param t     the other Observable that emits items to compare.
                \param pred  the function that implements comparison of two values.

                \return  Observable that emits true only if both sequences terminate normally after emitting the same sequence of items in the same order; otherwise it will emit false.

                \sample
                \snippet sequence_equal.cpp sequence_equal sample
                \snippet output.txt sequence_equal sample
                */
                template<class OtherSource, class BinaryPredicate>
                auto sequence_equal(OtherSource&& t, BinaryPredicate&& pred) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> typename std::enable_if<is_observable<OtherSource>::value && !is_coordination<BinaryPredicate>::value,
                        observable<bool, rxo::detail::sequence_equal<T, this_type, OtherSource, BinaryPredicate, identity_one_worker>>>::type
                        /// \endcond
                {
                        return  observable<bool, rxo::detail::sequence_equal<T, this_type, OtherSource, BinaryPredicate, identity_one_worker>>(
                                        rxo::detail::sequence_equal<T, this_type, OtherSource, BinaryPredicate, identity_one_worker>(*this, std::forward<OtherSource>(t), std::forward<BinaryPredicate>(pred), identity_one_worker(rxsc::make_current_thread())));
                }

                /*! Determine whether two Observables emit the same sequence of items.

                \tparam OtherSource   the type of the other observable.
                \tparam Coordination  the type of the scheduler.

                \param t  the other Observable that emits items to compare.
                \param cn the scheduler.

                \return  Observable that emits true only if both sequences terminate normally after emitting the same sequence of items in the same order; otherwise it will emit false.

                \sample
                \snippet sequence_equal.cpp sequence_equal sample
                \snippet output.txt sequence_equal sample
                */
                template<class OtherSource, class Coordination>
                auto sequence_equal(OtherSource&& t, Coordination&& cn) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> typename std::enable_if<is_observable<OtherSource>::value && is_coordination<Coordination>::value,
                        observable<bool, rxo::detail::sequence_equal<T, this_type, OtherSource, rxu::equal_to<>, Coordination>>>::type
                        /// \endcond
                {
                        return  observable<bool, rxo::detail::sequence_equal<T, this_type, OtherSource, rxu::equal_to<>, Coordination>>(
                                        rxo::detail::sequence_equal<T, this_type, OtherSource, rxu::equal_to<>, Coordination>(*this, std::forward<OtherSource>(t), rxu::equal_to<>(), std::forward<Coordination>(cn)));
```

```
        }

        /*! Determine whether two Observables emit the same sequence of items.

        \tparam OtherSource  the type of the other observable.

        \param t  the other Observable that emits items to compare.

        \return  Observable that emits true only if both sequences terminate normally after emitting the same sequence of items in the same order; otherwise it will emit false.

        \sample
        \snippet sequence_equal.cpp sequence_equal sample
        \snippet output.txt sequence_equal sample
        */
        template<class OtherSource>
        auto sequence_equal(OtherSource&& t) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> typename std::enable_if<is_observable<OtherSource>::value,
            observable<bool, rxo::detail::sequence_equal<T, this_type, OtherSource, rxu::equal_to<>, identity_one_worker>>>::type
            /// \endcond
        {
            return  observable<bool, rxo::detail::sequence_equal<T, this_type, OtherSource, rxu::equal_to<>, identity_one_worker>>(
                rxo::detail::sequence_equal<T, this_type, OtherSource, rxu::equal_to<>, identity_one_worker>(*this, std::forward<OtherSource>(t), rxu::equal_to<>(), identity_one_worker(rxsc::make_current_thread())));
        }

        /*! inspect calls to on_next, on_error and on_completed.

        \tparam MakeObserverArgN...  these args are passed to make_observer

        \param an  these args are passed to make_observer.

        \return  Observable that emits the same items as the source observable to both the subscriber and the observer.

        \note If an on_error method is not supplied the observer will ignore errors rather than call std::terminate()

        \sample
        \snippet tap.cpp tap sample
        \snippet output.txt tap sample

        If the source observable generates an error, the observer passed to tap is called:
        \snippet tap.cpp error tap sample
        \snippet output.txt error tap sample
        */
        template<class... MakeObserverArgN>
        auto tap(MakeObserverArgN&&... an) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS lift<T>(rxo::detail::tap<T, std::tuple<MakeObserverArgN...>>(std::make_tuple(std::forward<MakeObserverArgN>(an)...))))
            /// \endcond
        {
            return                  lift<T>(rxo::detail::tap<T, std::tuple<MakeObserverArgN...>>(std::make_tuple(std::forward<MakeObserverArgN>(an)...)));
        }

        /*! Returns an observable that emits indications of the amount of time lapsed between consecutive emissions of the source observable.
        The first emission from this new Observable indicates the amount of time lapsed between the time when the observer subscribed to the Observable and the time when the source Observable emitted its first item.

        \tparam Coordination  the type of the scheduler

        \param coordination  the scheduler for itme intervals

        \return  Observable that emits a time_duration to indicate the amount of time lapsed between pairs of emissions.

        \sample
        \snippet time_interval.cpp time_interval sample
        \snippet output.txt time_interval sample
        */
        template<class Coordination>
        auto time_interval(Coordination coordination) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS lift<rxsc::scheduler::clock_type::time_point::duration>(rxo::detail::time_interval<T, Coordination>{coordination}))
            /// \endcond
        {
            return              lift<rxsc::scheduler::clock_type::time_point::duration>(rxo::detail::time_interval<T, Coordination>{coordination});
```

```
        }

        /*! Returns an observable that emits indications of the amount of time lapsed between consecutive emissions of the source observable.
        The first emission from this new Observable indicates the amount of time lapsed between the time when the observer subscribed to the Observable and the time when the source Observable emitted its first item.

        \return  Observable that emits a time_duration to indicate the amount of time lapsed between pairs of emissions.

        \sample
        \snippet time_interval.cpp time_interval sample
        \snippet output.txt time_interval sample
        */
        template<class... AN>
        auto time_interval(AN**...) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS lift<rxsc::scheduler::clock_type::time_point::duration>(rxo::detail::time_interval<T, identity_one_worker>{identity_current_thread()}))
            /// \endcond
        {
            return lift<rxsc::scheduler::clock_type::time_point::duration>(rxo::detail::time_interval<T, identity_one_worker>{identity_current_thread()});
            static_assert(sizeof...(AN) == 0, "time_interval() was passed too many arguments.");
        }

        /*! Return an observable that terminates with timeout_error if a particular timespan has passed without emitting another item from the source observable.

        \tparam Duration      the type of time interval
        \tparam Coordination  the type of the scheduler

        \param period       the period of time wait for another item from the source observable.
        \param coordination  the scheduler to manage timeout for each event

        \return  Observable that terminates with an error if a particular timespan has passed without emitting another item from the source observable.

        \sample
        \snippet timeout.cpp timeout sample
        \snippet output.txt timeout sample
        */
        template<class Duration, class Coordination>
        auto timeout(Duration period, Coordination coordination) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS lift<T>(rxo::detail::timeout<T, Duration, Coordination>(period, coordination)))
            /// \endcond
        {
            return lift<T>(rxo::detail::timeout<T, Duration, Coordination>(period, coordination));
        }

        /*! Return an observable that terminates with timeout_error if a particular timespan has passed without emitting another item from the source observable.

        \tparam Duration      the type of time interval

        \param period       the period of time wait for another item from the source observable.

        \return  Observable that terminates with an error if a particular timespan has passed without emitting another item from the source observable.

        \sample
        \snippet timeout.cpp timeout sample
        \snippet output.txt timeout sample
        */
        template<class Duration>
        auto timeout(Duration period) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS lift<T>(rxo::detail::timeout<T, Duration, identity_one_worker>(period, identity_current_thread())))
            /// \endcond
        {
            return lift<T>(rxo::detail::timeout<T, Duration, identity_one_worker>(period, identity_current_thread()));
        }

        /*! Returns an observable that attaches a timestamp to each item emitted by the source observable indicating when it was emitted.

        \tparam Coordination  the type of the scheduler

        \param coordination  the scheduler to manage timeout for each event
```

```
                \return  Observable that emits a pair: { item emitted by the source observable, time_point representing the current value of
the clock }.

                \sample
                \snippet timestamp.cpp timestamp sample
                \snippet output.txt timestamp sample
                */
                template<class Coordination>
                auto timestamp(Coordination coordination) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS lift<std::pair<T,
rxsc::scheduler::clock_type::time_point>>(rxo::detail::timestamp<T, Coordination>{coordination}))
                        /// \endcond
                {
                        return          lift<std::pair<T, rxsc::scheduler::clock_type::time_point>>(rxo::detail::timestamp<T,
Coordination>{coordination});
                }

                /*! Returns an observable that attaches a timestamp to each item emitted by the source observable indicating when it was
emitted.

                \tparam ClockType    the type of the clock to return a time_point.

                \return  Observable that emits a pair: { item emitted by the source observable, time_point representing the current value of
the clock }.

                \sample
                \snippet timestamp.cpp timestamp sample
                \snippet output.txt timestamp sample
                */
                template<class... AN>
                auto timestamp(AN**...) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS lift<std::pair<T,
rxsc::scheduler::clock_type::time_point>>(rxo::detail::timestamp<T, identity_one_worker>{identity_current_thread()}))
                        /// \endcond
                {
                        return          lift<std::pair<T, rxsc::scheduler::clock_type::time_point>>(rxo::detail::timestamp<T,
identity_one_worker>{identity_current_thread()});
                        static_assert(sizeof...(AN) == 0, "timestamp() was passed too many arguments.");
                }

                /*! Add a new action at the end of the new observable that is returned.

                \tparam LastCall  the type of the action function

                \param lc  the action function

                \return  Observable that emits the same items as the source observable, then invokes the given action.

                \sample
                \snippet finally.cpp finally sample
                \snippet output.txt finally sample

                If the source observable generates an error, the final action is still being called:
                \snippet finally.cpp error finally sample
                \snippet output.txt error finally sample
                */
                template<class LastCall>
                auto finally(LastCall lc) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS lift<T>(rxo::detail::finally<T, LastCall>(std::move(lc))))
                        /// \endcond
                {
                        return          lift<T>(rxo::detail::finally<T, LastCall>(std::move(lc)));
                }

                /*! If an error occurs, take the result from the Selector and subscribe to that instead.

                \tparam Selector  the actual type of a function of the form `observable<T>(std::exception_ptr)`

                \param s  the function of the form `observable<T>(std::exception_ptr)`

                \return  Observable that emits the items from the source observable and switches to a new observable on error.

                \sample
                \snippet on_error_resume_next.cpp on_error_resume_next sample
                \snippet output.txt on_error_resume_next sample
                */
                template<class Selector>
                auto on_error_resume_next(Selector s) const
                        /// \cond SHOW_SERVICE_MEMBERS
```

```
                -> decltype(EXPLICIT_THIS lift<rxu::value_type_t<rxo::detail::on_error_resume_next<T,
Selector>>>(rxo::detail::on_error_resume_next<T, Selector>(std::move(s))))
                /// \endcond
            {
                return lift<rxu::value_type_t<rxo::detail::on_error_resume_next<T,
Selector>>>(rxo::detail::on_error_resume_next<T, Selector>(std::move(s)));
            }

            /*! For each item from this observable use Selector to produce an item to emit from the new observable that is returned.

            \tparam Selector  the type of the transforming function

            \param s  the selector function

            \return  Observable that emits the items from the source observable, transformed by the specified function.

            \sample
            \snippet map.cpp map sample
            \snippet output.txt map sample
            */
            template<class Selector>
            auto map(Selector s) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(EXPLICIT_THIS lift<rxu::value_type_t<rxo::detail::map<T, Selector>>>(rxo::detail::map<T,
Selector>(std::move(s))))
                /// \endcond
            {
                return lift<rxu::value_type_t<rxo::detail::map<T, Selector>>>(rxo::detail::map<T,
Selector>(std::move(s)));
            }

            /*! @copydoc rx-debounce.hpp
            */
            template<class... AN>
            auto debounce(AN&&... an) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(observable_member(debounce_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
                /// \endcond
            {
                return  observable_member(debounce_tag{}, *this, std::forward<AN>(an)...);
            }

            /*! @copydoc rx-delay.hpp
            */
            template<class... AN>
            auto delay(AN&&... an) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(observable_member(delay_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
                /// \endcond
            {
                return  observable_member(delay_tag{}, *this, std::forward<AN>(an)...);
            }

            /*! @copydoc rx-distinct.hpp
            */
            template<class... AN>
            auto distinct(AN&&... an) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(observable_member(distinct_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
                /// \endcond
            {
                return  observable_member(distinct_tag{}, *this, std::forward<AN>(an)...);
            }

            /*! @copydoc rx-distinct_until_changed.hpp
            */
            template<class... AN>
            auto distinct_until_changed(AN&&... an) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(observable_member(distinct_until_changed_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
                /// \endcond
            {
                return  observable_member(distinct_until_changed_tag{}, *this, std::forward<AN>(an)...);
            }

            /*! @copydoc rx-element_at.hpp
            */
            template<class... AN>
            auto element_at(AN&&... an) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(observable_member(element_at_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
                /// \endcond
```

```
        {
            return  observable_member(element_at_tag{}, *this, std::forward<AN>(an)...);
        }

        /*! Return an observable that emits connected, non-overlapping windows, each containing at most count items from the
source observable.

        \param count  the maximum size of each window before it should be completed

        \return  Observable that emits connected, non-overlapping windows, each containing at most count items from the source
observable.

        \sample
        \snippet window.cpp window count sample
        \snippet output.txt window count sample
        */
        template<class... AN>
        auto window(int count, AN**...) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS lift<observable<T>>(rxo::detail::window<T>(count, count)))
            /// \endcond
        {
            return              lift<observable<T>>(rxo::detail::window<T>(count, count));
            static_assert(sizeof...(AN) == 0, "window(count) was passed too many arguments.");
        }

        /*! Return an observable that emits windows every skip items containing at most count items from the source observable.

        \param count  the maximum size of each window before it should be completed
        \param skip   how many items need to be skipped before starting a new window

        \return  Observable that emits windows every skip items containing at most count items from the source observable.

        \sample
        \snippet window.cpp window count+skip sample
        \snippet output.txt window count+skip sample
        */
        template<class... AN>
        auto window(int count, int skip, AN**...) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS lift<observable<T>>(rxo::detail::window<T>(count, skip)))
            /// \endcond
        {
            return              lift<observable<T>>(rxo::detail::window<T>(count, skip));
            static_assert(sizeof...(AN) == 0, "window(count, skip) was passed too many arguments.");
        }

        /*! Return an observable that emits observables every skip time interval and collects items from this observable for period of
time into each produced observable, on the specified scheduler.

        \tparam Duration      the type of time intervals
        \tparam Coordination  the type of the scheduler

        \param period       the period of time each window collects items before it is completed
        \param skip         the period of time after which a new window will be created
        \param coordination  the scheduler for the windows

        \return  Observable that emits observables every skip time interval and collect items from this observable for period of time
into each produced observable.

        \sample
        \snippet window.cpp window period+skip+coordination sample
        \snippet output.txt window period+skip+coordination sample
        */
        template<class Duration, class Coordination>
        auto window_with_time(Duration period, Duration skip, Coordination coordination) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS lift<observable<T>>(rxo::detail::window_with_time<T, Duration,
Coordination>(period, skip, coordination)))
            /// \endcond
        {
            return              lift<observable<T>>(rxo::detail::window_with_time<T, Duration, Coordination>(period, skip,
coordination));
        }

        /*! Return an observable that emits observables every skip time interval and collects items from this observable for period of
time into each produced observable.

        \tparam Duration  the type of time intervals

        \param period  the period of time each window collects items before it is completed
        \param skip    the period of time after which a new window will be created
```

```
        \return  Observable that emits observables every skip time interval and collect items from this observable for period of time into each produced observable.

        \sample
        \snippet window.cpp window period+skip sample
        \snippet output.txt window period+skip sample
        */
        template<class Duration>
        auto window_with_time(Duration period, Duration skip) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS lift<observable<T>>(rxo::detail::window_with_time<T, Duration, identity_one_worker>(period, skip, identity_current_thread())))
            /// \endcond
        {
            return lift<observable<T>>(rxo::detail::window_with_time<T, Duration, identity_one_worker>(period, skip, identity_current_thread()));
        }

        /*! Return an observable that emits observables every period time interval and collects items from this observable for period of time into each produced observable, on the specified scheduler.

        \tparam Duration      the type of time intervals
        \tparam Coordination  the type of the scheduler

        \param period       the period of time each window collects items before it is completed and replaced with a new window
        \param coordination  the scheduler for the windows

        \return  Observable that emits observables every period time interval and collect items from this observable for period of time into each produced observable.

        \sample
        \snippet window.cpp window period+coordination sample
        \snippet output.txt window period+coordination sample
        */
        template<class Duration, class Coordination, class Reqiures = typename rxu::types_checked_from<typename Coordination::coordination_tag>::type>
        auto window_with_time(Duration period, Coordination coordination) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS lift<observable<T>>(rxo::detail::window_with_time<T, Duration, Coordination>(period, period, coordination)))
            /// \endcond
        {
            return lift<observable<T>>(rxo::detail::window_with_time<T, Duration, Coordination>(period, period, coordination));
        }

        /*! Return an observable that emits connected, non-overlapping windows represending items emitted by the source observable during fixed, consecutive durations.

        \tparam Duration  the type of time intervals

        \param period  the period of time each window collects items before it is completed and replaced with a new window

        \return  Observable that emits connected, non-overlapping windows represending items emitted by the source observable during fixed, consecutive durations.

        \sample
        \snippet window.cpp window period sample
        \snippet output.txt window period sample
        */
        template<class Duration>
        auto window_with_time(Duration period) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS lift<observable<T>>(rxo::detail::window_with_time<T, Duration, identity_one_worker>(period, period, identity_current_thread())))
            /// \endcond
        {
            return lift<observable<T>>(rxo::detail::window_with_time<T, Duration, identity_one_worker>(period, period, identity_current_thread()));
        }

        /*! Return an observable that emits connected, non-overlapping windows of items from the source observable that were emitted during a fixed duration of time or when the window has reached maximum capacity (whichever occurs first), on the specified scheduler.

        \tparam Duration      the type of time intervals
        \tparam Coordination  the type of the scheduler

        \param period       the period of time each window collects items before it is completed and replaced with a new window
        \param count        the maximum size of each window before it is completed and new window is created
        \param coordination  the scheduler for the windows
```

```
            \return  Observable that emits connected, non-overlapping windows of items from the source observable that were emitted during a fixed duration of time or when the window has reached maximum capacity (whichever occurs first).

            \sample
            \snippet window.cpp window period+count+coordination sample
            \snippet output.txt window period+count+coordination sample
            */
            template<class Duration, class Coordination>
            auto window_with_time_or_count(Duration period, int count, Coordination coordination) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(EXPLICIT_THIS lift<observable<T>>(rxo::detail::window_with_time_or_count<T, Duration, Coordination>(period, count, coordination)))
                /// \endcond
            {
                return      lift<observable<T>>(rxo::detail::window_with_time_or_count<T, Duration, Coordination>(period, count, coordination));
            }

            /*! Return an observable that emits connected, non-overlapping windows of items from the source observable that were emitted during a fixed duration of time or when the window has reached maximum capacity (whichever occurs first).

            \tparam Duration  the type of time intervals

            \param period  the period of time each window collects items before it is completed and replaced with a new window
            \param count   the maximum size of each window before it is completed and new window is created

            \return  Observable that emits connected, non-overlapping windows of items from the source observable that were emitted during a fixed duration of time or when the window has reached maximum capacity (whichever occurs first).

            \sample
            \snippet window.cpp window period+count sample
            \snippet output.txt window period+count sample
            */
            template<class Duration>
            auto window_with_time_or_count(Duration period, int count) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(EXPLICIT_THIS lift<observable<T>>(rxo::detail::window_with_time_or_count<T, Duration, identity_one_worker>(period, count, identity_current_thread())))
                /// \endcond
            {
                return      lift<observable<T>>(rxo::detail::window_with_time_or_count<T, Duration, identity_one_worker>(period, count, identity_current_thread()));
            }

            /*! Return an observable that emits observables every period time interval and collects items from this observable for period of time into each produced observable, on the specified scheduler.

            \tparam Openings       observable<OT>
            \tparam ClosingSelector a function of type observable<CT>(OT)
            \tparam Coordination    the type of the scheduler

            \param opens        each value from this observable opens a new window.
            \param closes       this function is called for each opened window and returns an observable. the first value from the returned observable will close the window
            \param coordination  the scheduler for the windows

            \return  Observable that emits an observable for each opened window.

            \sample
            \snippet window.cpp window toggle+coordination sample
            \snippet output.txt window toggle+coordination sample
            */
            template<class Openings, class ClosingSelector, class Coordination, class Reqiures = typename rxu::types_checked_from<typename Coordination::coordination_tag>::type>
            auto window_toggle(Openings opens, ClosingSelector closes, Coordination coordination) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(EXPLICIT_THIS lift<observable<T>>(rxo::detail::window_toggle<T, Openings, ClosingSelector, Coordination>(opens, closes, coordination)))
                /// \endcond
            {
                return      lift<observable<T>>(rxo::detail::window_toggle<T, Openings, ClosingSelector, Coordination>(opens, closes, coordination));
            }

            /*! Return an observable that emits connected, non-overlapping windows represending items emitted by the source observable during fixed, consecutive durations.

            \tparam Openings       observable<OT>
            \tparam ClosingSelector a function of type observable<CT>(OT)

            \param opens        each value from this observable opens a new window.
```

```
                \param closes       this function is called for each opened window and returns an observable. the first value from the returned observable will close the window

                \return  Observable that emits an observable for each opened window.

                \sample
                \snippet window.cpp window toggle sample
                \snippet output.txt window toggle sample
                */
                template<class Openings, class ClosingSelector>
                auto window_toggle(Openings opens, ClosingSelector closes) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS lift<observable<T>>(rxo::detail::window_toggle<T, Openings, ClosingSelector, identity_one_worker>(opens, closes, identity_current_thread())))
                        /// \endcond
                {
                        return                  lift<observable<T>>(rxo::detail::window_toggle<T, Openings, ClosingSelector, identity_one_worker>(opens, closes, identity_current_thread()));
                }

                /*! Return an observable that emits connected, non-overlapping buffer, each containing at most count items from the source observable.

                \param count  the maximum size of each buffer before it should be emitted

                \return  Observable that emits connected, non-overlapping buffers, each containing at most count items from the source observable.

                \sample
                \snippet buffer.cpp buffer count sample
                \snippet output.txt buffer count sample
                */
                template<class... AN>
                auto buffer(int count, AN**...) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS lift_if<std::vector<T>>(rxo::detail::buffer_count<T>(count, count)))
                        /// \endcond
                {
                        return                  lift_if<std::vector<T>>(rxo::detail::buffer_count<T>(count, count));
                        static_assert(sizeof...(AN) == 0, "buffer(count) was passed too many arguments.");
                }

                /*! Return an observable that emits buffers every skip items containing at most count items from the source observable.

                \param count  the maximum size of each buffers before it should be emitted
                \param skip   how many items need to be skipped before starting a new buffers

                \return  Observable that emits buffers every skip items containing at most count items from the source observable.

                \sample
                \snippet buffer.cpp buffer count+skip sample
                \snippet output.txt buffer count+skip sample
                */
                template<class... AN>
                auto buffer(int count, int skip, AN**...) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS lift_if<std::vector<T>>(rxo::detail::buffer_count<T>(count, skip)))
                        /// \endcond
                {
                        return                  lift_if<std::vector<T>>(rxo::detail::buffer_count<T>(count, skip));
                        static_assert(sizeof...(AN) == 0, "buffer(count, skip) was passed too many arguments.");
                }

                /*! Return an observable that emits buffers every skip time interval and collects items from this observable for period of time into each produced buffer, on the specified scheduler.

                \tparam Coordination  the type of the scheduler

                \param period       the period of time each buffer collects items before it is emitted
                \param skip         the period of time after which a new buffer will be created
                \param coordination  the scheduler for the buffers

                \return  Observable that emits buffers every skip time interval and collect items from this observable for period of time into each produced buffer.

                \sample
                \snippet buffer.cpp buffer period+skip+coordination sample
                \snippet output.txt buffer period+skip+coordination sample
                */
                template<class Coordination>
                auto buffer_with_time(rxsc::scheduler::clock_type::duration period, rxsc::scheduler::clock_type::duration skip, Coordination coordination) const
```

```
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(EXPLICIT_THIS lift_if<std::vector<T>>(rxo::detail::buffer_with_time<T,
rxsc::scheduler::clock_type::duration, Coordination>(period, skip, coordination)))
                /// \endcond
        {
                return          lift_if<std::vector<T>>(rxo::detail::buffer_with_time<T,
rxsc::scheduler::clock_type::duration, Coordination>(period, skip, coordination));
        }

        /*! Return an observable that emits buffers every skip time interval and collects items from this observable for period of
time into each produced buffer.

        \param period       the period of time each buffer collects items before it is emitted
        \param skip         the period of time after which a new buffer will be created

        \return  Observable that emits buffers every skip time interval and collect items from this observable for period of time into
each produced buffer.

        \sample
        \snippet buffer.cpp buffer period+skip sample
        \snippet output.txt buffer period+skip sample

        Overlapping buffers are allowed:
        \snippet buffer.cpp buffer period+skip overlapping sample
        \snippet output.txt buffer period+skip overlapping sample

        If no items are emitted, an empty buffer is returned:
        \snippet buffer.cpp buffer period+skip empty sample
        \snippet output.txt buffer period+skip empty sample
        */
        template<class Duration>
        auto buffer_with_time(Duration period, Duration skip) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(EXPLICIT_THIS lift_if<std::vector<T>>(rxo::detail::buffer_with_time<T, Duration,
identity_one_worker>(period, skip, identity_current_thread())))
                /// \endcond
        {
                return          lift_if<std::vector<T>>(rxo::detail::buffer_with_time<T, Duration,
identity_one_worker>(period, skip, identity_current_thread()));
        }

        /*! Return an observable that emits buffers every period time interval and collects items from this observable for period of
time into each produced buffer, on the specified scheduler.

        \tparam Coordination  the type of the scheduler

        \param period       the period of time each buffer collects items before it is emitted and replaced with a new buffer
        \param coordination  the scheduler for the buffers

        \return  Observable that emits buffers every period time interval and collect items from this observable for period of time
into each produced buffer.

        \sample
        \snippet buffer.cpp buffer period+coordination sample
        \snippet output.txt buffer period+coordination sample
        */
        template<class Coordination,
        class Requires = typename std::enable_if<is_coordination<Coordination>::value, rxu::types_checked>::type>
                auto buffer_with_time(rxsc::scheduler::clock_type::duration period, Coordination coordination) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(EXPLICIT_THIS lift_if<std::vector<T>>(rxo::detail::buffer_with_time<T,
rxsc::scheduler::clock_type::duration, Coordination>(period, period, coordination)))
                /// \endcond
        {
                        return          lift_if<std::vector<T>>(rxo::detail::buffer_with_time<T,
rxsc::scheduler::clock_type::duration, Coordination>(period, period, coordination));
                }

        /*! Return an observable that emits buffers every period time interval and collects items from this observable for period of
time into each produced buffer.

        \param period  the period of time each buffer collects items before it is emitted and replaced with a new buffer

        \return  Observable that emits buffers every period time interval and collect items from this observable for period of time
into each produced buffer.

        \sample
        \snippet buffer.cpp buffer period sample
        \snippet output.txt buffer period sample
        */
        template<class Duration>
        auto buffer_with_time(Duration period) const
```

```
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(EXPLICIT_THIS lift_if<std::vector<T>>(rxo::detail::buffer_with_time<T, Duration,
identity_one_worker>(period, period, identity_current_thread())))
                /// \endcond
        {
                return          lift_if<std::vector<T>>(rxo::detail::buffer_with_time<T, Duration,
identity_one_worker>(period, period, identity_current_thread()));
        }

        /*! Return an observable that emits connected, non-overlapping buffers of items from the source observable that were
emitted during a fixed duration of time or when the buffer has reached maximum capacity (whichever occurs first), on the specified scheduler.

        \tparam Coordination  the type of the scheduler

        \param period       the period of time each buffer collects items before it is emitted and replaced with a new buffer
        \param count        the maximum size of each buffer before it is emitted and new buffer is created
        \param coordination  the scheduler for the buffers

        \return  Observable that emits connected, non-overlapping buffers of items from the source observable that were emitted
during a fixed duration of time or when the buffer has reached maximum capacity (whichever occurs first).

        \sample
        \snippet buffer.cpp buffer period+count+coordination sample
        \snippet output.txt buffer period+count+coordination sample
        */
        template<class Coordination>
        auto buffer_with_time_or_count(rxsc::scheduler::clock_type::duration period, int count, Coordination coordination) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(EXPLICIT_THIS lift_if<std::vector<T>>(rxo::detail::buffer_with_time_or_count<T,
rxsc::scheduler::clock_type::duration, Coordination>(period, count, coordination)))
                /// \endcond
        {
                return          lift_if<std::vector<T>>(rxo::detail::buffer_with_time_or_count<T,
rxsc::scheduler::clock_type::duration, Coordination>(period, count, coordination));
        }

        /*! Return an observable that emits connected, non-overlapping buffers of items from the source observable that were
emitted during a fixed duration of time or when the buffer has reached maximum capacity (whichever occurs first).

        \param period       the period of time each buffer collects items before it is emitted and replaced with a new buffer
        \param count        the maximum size of each buffer before it is emitted and new buffer is created

        \return  Observable that emits connected, non-overlapping buffers of items from the source observable that were emitted
during a fixed duration of time or when the buffer has reached maximum capacity (whichever occurs first).

        \sample
        \snippet buffer.cpp buffer period+count sample
        \snippet output.txt buffer period+count sample
        */
        template<class Duration>
        auto buffer_with_time_or_count(Duration period, int count) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(EXPLICIT_THIS lift_if<std::vector<T>>(rxo::detail::buffer_with_time_or_count<T, Duration,
identity_one_worker>(period, count, identity_current_thread())))
                /// \endcond
        {
                return          lift_if<std::vector<T>>(rxo::detail::buffer_with_time_or_count<T, Duration,
identity_one_worker>(period, count, identity_current_thread()));
        }

        /// \cond SHOW_SERVICE_MEMBERS
        template<class Coordination>
        struct defer_switch_on_next : public defer_observable<
                is_observable<value_type>,
                this_type,
                rxo::detail::switch_on_next, value_type, observable<value_type>, Coordination>
        {
        };
        /// \endcond

        /*! Return observable that emits the items emitted by the observable most recently emitted by the source observable.

        \return  Observable that emits the items emitted by the observable most recently emitted by the source observable.

        \note All sources must be synchronized! This means that calls across all the subscribers must be serial.

        \sample
        \snippet switch_on_next.cpp switch_on_next sample
        \snippet output.txt switch_on_next sample
        */
        template<class... AN>
        auto switch_on_next(AN**...) const
```

```
                /// \cond SHOW_SERVICE_MEMBERS
                -> typename defer_switch_on_next<identity_one_worker>::observable_type
                /// \endcond
        {
                return      defer_switch_on_next<identity_one_worker>::make(*this, *this, identity_current_thread());
                static_assert(sizeof...(AN) == 0, "switch_on_next() was passed too many arguments.");
        }

        /*! Return observable that emits the items emitted by the observable most recently emitted by the source observable, on the
specified scheduler.

        \tparam Coordination  the type of the scheduler

        \param cn  the scheduler to synchronize sources from different contexts

        \return  Observable that emits the items emitted by the observable most recently emitted by the source observable.

        \sample
        \snippet switch_on_next.cpp threaded switch_on_next sample
        \snippet output.txt threaded switch_on_next sample
        */
        template<class Coordination>
        auto switch_on_next(Coordination cn) const
                /// \cond SHOW_SERVICE_MEMBERS
                ->  typename std::enable_if<
                defer_switch_on_next<Coordination>::value,
                typename    defer_switch_on_next<Coordination>::observable_type>::type
                /// \endcond
        {
                return          defer_switch_on_next<Coordination>::make(*this, *this, std::move(cn));
        }

        /// \cond SHOW_SERVICE_MEMBERS
        template<class Coordination>
        struct defer_merge : public defer_observable<
                is_observable<value_type>,
                this_type,
                rxo::detail::merge, value_type, observable<value_type>, Coordination>
        {
        };
        /// \endcond

        /*! For each item from this observable subscribe.
        For each item from all of the nested observables deliver from the new observable that is returned.

        \return  Observable that emits items that are the result of flattening the observables emitted by the source observable.

        \note All sources must be synchronized! This means that calls across all the subscribers must be serial.

        \sample
        \snippet merge.cpp implicit merge sample
        \snippet output.txt implicit merge sample
        */
        template<class... AN>
        auto merge(AN**...) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> typename defer_merge<identity_one_worker>::observable_type
                /// \endcond
        {
                return      defer_merge<identity_one_worker>::make(*this, *this, identity_current_thread());
                static_assert(sizeof...(AN) == 0, "merge() was passed too many arguments.");
        }

        /*! For each item from this observable subscribe.
        For each item from all of the nested observables deliver from the new observable that is returned.

        \tparam Coordination  the type of the scheduler

        \param  cn  the scheduler to synchronize sources from different contexts.

        \return  Observable that emits items that are the result of flattening the observables emitted by the source observable.

        \sample
        \snippet merge.cpp threaded implicit merge sample
        \snippet output.txt threaded implicit merge sample
        */
        template<class Coordination>
        auto merge(Coordination cn) const
                /// \cond SHOW_SERVICE_MEMBERS
                ->  typename std::enable_if<
                defer_merge<Coordination>::value,
                typename    defer_merge<Coordination>::observable_type>::type
```

```
                /// \endcond
        {
                return  defer_merge<Coordination>::make(*this, *this, std::move(cn));
        }

        /// \cond SHOW_SERVICE_MEMBERS
        template<class Coordination, class Value0>
        struct defer_merge_from : public defer_observable<
                rxu::all_true<
                is_coordination<Coordination>::value,
                is_observable<Value0>::value>,
                this_type,
                rxo::detail::merge, observable<value_type>, observable<observable<value_type>>, Coordination>
        {
        };
        /// \endcond

        /*! For each given observable subscribe.
        For each emitted item deliver from the new observable that is returned.

        \tparam Value0  ...
        \tparam ValueN  types of source observables

        \param  v0  ...
        \param  vn  source observables

        \return  Observable that emits items that are the result of flattening the observables emitted by the source observable.

        \note All sources must be synchronized! This means that calls across all the subscribers must be serial.

        \sample
        \snippet merge.cpp merge sample
        \snippet output.txt merge sample
        */
        template<class Value0, class... ValueN>
        auto merge(Value0 v0, ValueN... vn) const
                /// \cond SHOW_SERVICE_MEMBERS
                ->  typename std::enable_if<
                defer_merge_from<identity_one_worker, Value0>::value,
                typename    defer_merge_from<identity_one_worker, Value0>::observable_type>::type
                /// \endcond
        {
                return  defer_merge_from<identity_one_worker, Value0>::make(*this, rxs::from(this->as_dynamic(),
v0.as_dynamic(), vn.as_dynamic()...), identity_current_thread());
        }

        /*! For each given observable subscribe.
        For each emitted item deliver from the new observable that is returned.

        \tparam Coordination  the type of the scheduler
        \tparam Value0        ...
        \tparam ValueN        types of source observables

        \param  cn  the scheduler to synchronize sources from different contexts.
        \param  v0  ...
        \param  vn  source observables

        \return  Observable that emits items that are the result of flattening the observables emitted by the source observable.

        \sample
        \snippet merge.cpp threaded merge sample
        \snippet output.txt threaded merge sample
        */
        template<class Coordination, class Value0, class... ValueN>
        auto merge(Coordination cn, Value0 v0, ValueN... vn) const
                /// \cond SHOW_SERVICE_MEMBERS
                ->  typename std::enable_if<
                defer_merge_from<Coordination, Value0>::value,
                typename    defer_merge_from<Coordination, Value0>::observable_type>::type
                /// \endcond
        {
                return  defer_merge_from<Coordination, Value0>::make(*this, rxs::from(this->as_dynamic(),
v0.as_dynamic(), vn.as_dynamic()...), std::move(cn));
        }

        /// \cond SHOW_SERVICE_MEMBERS
        template<class Coordination>
        struct defer_amb : public defer_observable<
                is_observable<value_type>,
                this_type,
                rxo::detail::amb, value_type, observable<value_type>, Coordination>
        {
```

```
        };
        /// \endcond

        /*! For each item from only the first of the nested observables deliver from the new observable that is returned.

        \return  Observable that emits the same sequence as whichever of the observables emitted from this observable that first emitted an item or sent a termination notification.

        \note All sources must be synchronized! This means that calls across all the subscribers must be serial.

        \sample
        \snippet amb.cpp implicit amb sample
        \snippet output.txt implicit amb sample
        */
        template<class... AN>
        auto amb(AN**...) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> typename defer_amb<identity_one_worker>::observable_type
            /// \endcond
        {
            return      defer_amb<identity_one_worker>::make(*this, *this, identity_current_thread());
            static_assert(sizeof...(AN) == 0, "amb() was passed too many arguments.");
        }

        /*! For each item from only the first of the nested observables deliver from the new observable that is returned, on the specified scheduler.

        \tparam Coordination  the type of the scheduler

        \param  cn  the scheduler to synchronize sources from different contexts.

        \return  Observable that emits the same sequence as whichever of the observables emitted from this observable that first emitted an item or sent a termination notification.

        \sample
        \snippet amb.cpp threaded implicit amb sample
        \snippet output.txt threaded implicit amb sample
        */
        template<class Coordination>
        auto amb(Coordination cn) const
            /// \cond SHOW_SERVICE_MEMBERS
            ->  typename std::enable_if<
            defer_amb<Coordination>::value,
            typename    defer_amb<Coordination>::observable_type>::type
            /// \endcond
        {
            return          defer_amb<Coordination>::make(*this, *this, std::move(cn));
        }

        /// \cond SHOW_SERVICE_MEMBERS
        template<class Coordination, class Value0>
        struct defer_amb_from : public defer_observable<
            rxu::all_true<
            is_coordination<Coordination>::value,
            is_observable<Value0>::value>,
            this_type,
            rxo::detail::amb, observable<value_type>, observable<observable<value_type>>, Coordination>
        {
        };
        /// \endcond

        /*! For each item from only the first of the given observables deliver from the new observable that is returned.

        \tparam Value0      ...
        \tparam ValueN      types of source observables

        \param  v0  ...
        \param  vn  source observables

        \return  Observable that emits the same sequence as whichever of the source observables first emitted an item or sent a termination notification.

        \note All sources must be synchronized! This means that calls across all the subscribers must be serial.

        \sample
        \snippet amb.cpp amb sample
        \snippet output.txt amb sample
        */
        template<class Value0, class... ValueN>
        auto amb(Value0 v0, ValueN... vn) const
            /// \cond SHOW_SERVICE_MEMBERS
            ->  typename std::enable_if<
```

```
                                        defer_amb_from<identity_one_worker, Value0>::value,
                                        typename    defer_amb_from<identity_one_worker, Value0>::observable_type>::type
                                        /// \endcond
                        {
                                        return          defer_amb_from<identity_one_worker, Value0>::make(*this, rxs::from(this->as_dynamic(),
v0.as_dynamic(), vn.as_dynamic()...), identity_current_thread());
                        }

                        /*! For each item from only the first of the given observables deliver from the new observable that is returned, on the
specified scheduler.

                        \tparam Coordination  the type of the scheduler
                        \tparam Value0        ...
                        \tparam ValueN        types of source observables

                        \param  cn  the scheduler to synchronize sources from different contexts.
                        \param  v0  ...
                        \param  vn  source observables

                        \return  Observable that emits the same sequence as whichever of the source observables first emitted an item or sent a
termination notification.

                        \sample
                        \snippet amb.cpp threaded amb sample
                        \snippet output.txt threaded amb sample
                        */
                        template<class Coordination, class Value0, class... ValueN>
                        auto amb(Coordination cn, Value0 v0, ValueN... vn) const
                                        /// \cond SHOW_SERVICE_MEMBERS
                                        ->  typename std::enable_if<
                                        defer_amb_from<Coordination, Value0>::value,
                                        typename    defer_amb_from<Coordination, Value0>::observable_type>::type
                                        /// \endcond
                        {
                                        return          defer_amb_from<Coordination, Value0>::make(*this, rxs::from(this->as_dynamic(),
v0.as_dynamic(), vn.as_dynamic()...), std::move(cn));
                        }

                        /*! For each item from this observable use the CollectionSelector to produce an observable and subscribe to that observable.
                        For each item from all of the produced observables use the ResultSelector to produce a value to emit from the new
observable that is returned.

                        \tparam CollectionSelector  the type of the observable producing function
                        \tparam ResultSelector      the type of the aggregation function

                        \param  s   a function that returns an observable for each item emitted by the source observable
                        \param  rs  a function that combines one item emitted by each of the source and collection observables and returns an item to
be emitted by the resulting observable

                        \return  Observable that emits the results of applying a function to a pair of values emitted by the source observable and the
collection observable.

                        Observables, produced by the CollectionSelector, are merged. There is another operator
rxcpp::observable<T,SourceType>::concat_map that works similar but concatenates the observables.

                        \sample
                        \snippet flat_map.cpp flat_map sample
                        \snippet output.txt flat_map sample
                        */
                        template<class CollectionSelector, class ResultSelector>
                        auto flat_map(CollectionSelector&& s, ResultSelector&& rs) const
                                        /// \cond SHOW_SERVICE_MEMBERS
                                        ->      observable<rxu::value_type_t<rxo::detail::flat_map<this_type, CollectionSelector, ResultSelector,
identity_one_worker>>, rxo::detail::flat_map<this_type, CollectionSelector, ResultSelector, identity_one_worker>>
                                        /// \endcond
                        {
                                        return  observable<rxu::value_type_t<rxo::detail::flat_map<this_type, CollectionSelector, ResultSelector,
identity_one_worker>>, rxo::detail::flat_map<this_type, CollectionSelector, ResultSelector, identity_one_worker>>(
                                                        rxo::detail::flat_map<this_type, CollectionSelector, ResultSelector, identity_one_worker>(*this,
std::forward<CollectionSelector>(s), std::forward<ResultSelector>(rs), identity_current_thread()));
                        }

                        /*! For each item from this observable use the CollectionSelector to produce an observable and subscribe to that observable.
                        For each item from all of the produced observables use the ResultSelector to produce a value to emit from the new
observable that is returned.

                        \tparam CollectionSelector  the type of the observable producing function
                        \tparam ResultSelector      the type of the aggregation function
                        \tparam Coordination        the type of the scheduler

                        \param  s   a function that returns an observable for each item emitted by the source observable
```

```
                \param  rs  a function that combines one item emitted by each of the source and collection observables and returns an item to be emitted by the resulting observable
                \param  cn  the scheduler to synchronize sources from different contexts.

                \return  Observable that emits the results of applying a function to a pair of values emitted by the source observable and the collection observable.

                Observables, produced by the CollectionSelector, are merged. There is another operator rxcpp::observable<T,SourceType>::concat_map that works similar but concatenates the observables.

                \sample
                \snippet flat_map.cpp threaded flat_map sample
                \snippet output.txt threaded flat_map sample
                */
                template<class CollectionSelector, class ResultSelector, class Coordination>
                auto flat_map(CollectionSelector&& s, ResultSelector&& rs, Coordination&& cn) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        ->      observable<rxu::value_type_t<rxo::detail::flat_map<this_type, CollectionSelector, ResultSelector, Coordination>>, rxo::detail::flat_map<this_type, CollectionSelector, ResultSelector, Coordination>>
                        /// \endcond
                {
                        return  observable<rxu::value_type_t<rxo::detail::flat_map<this_type, CollectionSelector, ResultSelector, Coordination>>, rxo::detail::flat_map<this_type, CollectionSelector, ResultSelector, Coordination>>(
                                        rxo::detail::flat_map<this_type, CollectionSelector, ResultSelector, Coordination>(*this, std::forward<CollectionSelector>(s), std::forward<ResultSelector>(rs), std::forward<Coordination>(cn)));
                }

                /// \cond SHOW_SERVICE_MEMBERS
                template<class Coordination>
                struct defer_concat : public defer_observable<
                        is_observable<value_type>,
                        this_type,
                        rxo::detail::concat, value_type, observable<value_type>, Coordination>
                {
                };
                /// \endcond

                /*! For each item from this observable subscribe to one at a time, in the order received.
                For each item from all of the nested observables deliver from the new observable that is returned.

                \return  Observable that emits the items emitted by each of the Observables emitted by the source observable, one after the other, without interleaving them.

                \note All sources must be synchronized! This means that calls across all the subscribers must be serial.

                \sample
                \snippet concat.cpp implicit concat sample
                \snippet output.txt implicit concat sample
                */
                template<class... AN>
                auto concat(AN**...) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> typename defer_concat<identity_one_worker>::observable_type
                        /// \endcond
                {
                        return      defer_concat<identity_one_worker>::make(*this, *this, identity_current_thread());
                        static_assert(sizeof...(AN) == 0, "concat() was passed too many arguments.");
                }

                /*! For each item from this observable subscribe to one at a time, in the order received.
                For each item from all of the nested observables deliver from the new observable that is returned.

                \tparam  Coordination  the type of the scheduler

                \param  cn  the scheduler to synchronize sources from different contexts.

                \return  Observable that emits the items emitted by each of the Observables emitted by the source observable, one after the other, without interleaving them.

                \note All sources must be synchronized! This means that calls across all the subscribers must be serial.

                \sample
                \snippet concat.cpp threaded implicit concat sample
                \snippet output.txt threaded implicit concat sample
                */
                template<class Coordination>
                auto concat(Coordination cn) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        ->  typename std::enable_if<
                        defer_concat<Coordination>::value,
                        typename    defer_concat<Coordination>::observable_type>::type
                        /// \endcond
```

```
                {
                        return  defer_concat<Coordination>::make(*this, *this, std::move(cn));
                }

                /// \cond SHOW_SERVICE_MEMBERS
                template<class Coordination, class Value0>
                struct defer_concat_from : public defer_observable<
                        rxu::all_true<
                        is_coordination<Coordination>::value,
                        is_observable<Value0>::value>,
                        this_type,
                        rxo::detail::concat, observable<value_type>, observable<observable<value_type>>, Coordination>
                {
                };
                /// \endcond

                /*! For each given observable subscribe to one at a time, in the order received.
                For each emitted item deliver from the new observable that is returned.

                \tparam Value0  ...
                \tparam ValueN  types of source observables

                \param  v0  ...
                \param  vn  source observables

                \return  Observable that emits items emitted by the source observables, one after the other, without interleaving them.

                \sample
                \snippet concat.cpp concat sample
                \snippet output.txt concat sample
                */
                template<class Value0, class... ValueN>
                auto concat(Value0 v0, ValueN... vn) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        ->  typename std::enable_if<
                        defer_concat_from<identity_one_worker, Value0>::value,
                        typename    defer_concat_from<identity_one_worker, Value0>::observable_type>::type
                        /// \endcond
                {
                        return  defer_concat_from<identity_one_worker, Value0>::make(*this, rxs::from(this->as_dynamic(),
v0.as_dynamic(), vn.as_dynamic()...), identity_current_thread());
                }

                /*! For each given observable subscribe to one at a time, in the order received.
                For each emitted item deliver from the new observable that is returned.

                \tparam Coordination  the type of the scheduler
                \tparam Value0        ...
                \tparam ValueN        types of source observables

                \param  cn  the scheduler to synchronize sources from different contexts.
                \param  v0  ...
                \param  vn  source observables

                \return  Observable that emits items emitted by the source observables, one after the other, without interleaving them.

                \sample
                \snippet concat.cpp threaded concat sample
                \snippet output.txt threaded concat sample
                */
                template<class Coordination, class Value0, class... ValueN>
                auto concat(Coordination cn, Value0 v0, ValueN... vn) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        ->  typename std::enable_if<
                        defer_concat_from<Coordination, Value0>::value,
                        typename    defer_concat_from<Coordination, Value0>::observable_type>::type
                        /// \endcond
                {
                        return  defer_concat_from<Coordination, Value0>::make(*this, rxs::from(this->as_dynamic(),
v0.as_dynamic(), vn.as_dynamic()...), std::move(cn));
                }

                /*! For each item from this observable use the CollectionSelector to produce an observable and subscribe to that observable.
                For each item from all of the produced observables use the ResultSelector to produce a value to emit from the new
observable that is returned.

                \tparam CollectionSelector  the type of the observable producing function
                \tparam ResultSelector      the type of the aggregation function

                \param  s   a function that returns an observable for each item emitted by the source observable
                \param  rs  a function that combines one item emitted by each of the source and collection observables and returns an item to
be emitted by the resulting observable
```

```
                \return  Observable that emits the results of applying a function to a pair of values emitted by the source observable and the collection observable.

                Observables, produced by the CollectionSelector, are concatenated. There is another operator rxcpp::observable<T,SourceType>::flat_map that works similar but merges the observables.

                \sample
                \snippet concat_map.cpp concat_map sample
                \snippet output.txt concat_map sample
                */
                template<class CollectionSelector, class ResultSelector>
                auto concat_map(CollectionSelector&& s, ResultSelector&& rs) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        ->      observable<rxu::value_type_t<rxo::detail::concat_map<this_type, CollectionSelector, ResultSelector, identity_one_worker>>, rxo::detail::concat_map<this_type, CollectionSelector, ResultSelector, identity_one_worker>>
                        /// \endcond
                {
                        return  observable<rxu::value_type_t<rxo::detail::concat_map<this_type, CollectionSelector, ResultSelector, identity_one_worker>>, rxo::detail::concat_map<this_type, CollectionSelector, ResultSelector, identity_one_worker>>(
                                        rxo::detail::concat_map<this_type, CollectionSelector, ResultSelector, identity_one_worker>(*this, std::forward<CollectionSelector>(s), std::forward<ResultSelector>(rs), identity_current_thread()));
                }

                /*! For each item from this observable use the CollectionSelector to produce an observable and subscribe to that observable.
                For each item from all of the produced observables use the ResultSelector to produce a value to emit from the new observable that is returned.

                \tparam CollectionSelector  the type of the observable producing function
                \tparam ResultSelector      the type of the aggregation function
                \tparam Coordination        the type of the scheduler

                \param  s   a function that returns an observable for each item emitted by the source observable
                \param  rs  a function that combines one item emitted by each of the source and collection observables and returns an item to be emitted by the resulting observable
                \param  cn  the scheduler to synchronize sources from different contexts.

                \return  Observable that emits the results of applying a function to a pair of values emitted by the source observable and the collection observable.

                Observables, produced by the CollectionSelector, are concatenated. There is another operator rxcpp::observable<T,SourceType>::flat_map that works similar but merges the observables.

                \sample
                \snippet concat_map.cpp threaded concat_map sample
                \snippet output.txt threaded concat_map sample
                */
                template<class CollectionSelector, class ResultSelector, class Coordination>
                auto concat_map(CollectionSelector&& s, ResultSelector&& rs, Coordination&& cn) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        ->      observable<rxu::value_type_t<rxo::detail::concat_map<this_type, CollectionSelector, ResultSelector, Coordination>>, rxo::detail::concat_map<this_type, CollectionSelector, ResultSelector, Coordination>>
                        /// \endcond
                {
                        return  observable<rxu::value_type_t<rxo::detail::concat_map<this_type, CollectionSelector, ResultSelector, Coordination>>, rxo::detail::concat_map<this_type, CollectionSelector, ResultSelector, Coordination>>(
                                        rxo::detail::concat_map<this_type, CollectionSelector, ResultSelector, Coordination>(*this, std::forward<CollectionSelector>(s), std::forward<ResultSelector>(rs), std::forward<Coordination>(cn)));
                }

                /*! @copydoc rx-with_latest_from.hpp
                */
                template<class... AN>
                auto with_latest_from(AN... an) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(observable_member(with_latest_from_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
                        /// \endcond
                {
                        return      observable_member(with_latest_from_tag{}, *this, std::forward<AN>(an)...);
                }


                /*! @copydoc rx-combine_latest.hpp
                */
                template<class... AN>
                auto combine_latest(AN... an) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(observable_member(combine_latest_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
                        /// \endcond
                {
                        return      observable_member(combine_latest_tag{}, *this, std::forward<AN>(an)...);
                }
```

```
        /*! @copydoc rx-zip.hpp
        */
        template<class... AN>
        auto zip(AN&&... an) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(observable_member(zip_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
            /// \endcond
        {
            return      observable_member(zip_tag{}, *this, std::forward<AN>(an)...);
        }

        /*! @copydoc rx-group_by.hpp
        */
        template<class... AN>
        inline auto group_by(AN&&... an) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(observable_member(group_by_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
            /// \endcond
        {
            return      observable_member(group_by_tag{}, *this, std::forward<AN>(an)...);
        }

        /*! @copydoc rx-ignore_elements.hpp
        */
        template<class... AN>
        auto ignore_elements(AN&&... an) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(observable_member(ignore_elements_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
            /// \endcond
        {
            return  observable_member(ignore_elements_tag{}, *this, std::forward<AN>(an)...);
        }

        /// \cond SHOW_SERVICE_MEMBERS
        /// multicast ->
        /// allows connections to the source to be independent of subscriptions
        ///
        template<class Subject>
        auto multicast(Subject sub) const
            ->      connectable_observable<T, rxo::detail::multicast<T, this_type, Subject>> {
            return  connectable_observable<T, rxo::detail::multicast<T, this_type, Subject>>(
                        rxo::detail::multicast<T, this_type, Subject>(*this, std::move(sub)));
        }
        /// \endcond

        /*! Turn a cold observable hot and allow connections to the source to be independent of subscriptions.

        \tparam  Coordination  the type of the scheduler

        \param  cn  a scheduler all values are queued and delivered on
        \param  cs  the subscription to control lifetime

        \return  rxcpp::connectable_observable that upon connection causes the source observable to emit items to its observers, on the specified scheduler.

        \sample
        \snippet publish.cpp publish_synchronized sample
        \snippet output.txt publish_synchronized sample
        */
        template<class Coordination>
        auto publish_synchronized(Coordination cn, composite_subscription cs = composite_subscription()) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS multicast(rxsub::synchronize<T, Coordination>(std::move(cn), cs)))
            /// \endcond
        {
            return                  multicast(rxsub::synchronize<T, Coordination>(std::move(cn), cs));
        }

        /*! Turn a cold observable hot and allow connections to the source to be independent of subscriptions.

        \return  rxcpp::connectable_observable that upon connection causes the source observable to emit items to its observers.

        \sample
        \snippet publish.cpp publish subject sample
        \snippet output.txt publish subject sample
        */
        template<class... AN>
        auto publish(AN**...) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS multicast(rxsub::subject<T>(composite_subscription())))
            /// \endcond
```

```
        {
            composite_subscription cs;
            return          multicast(rxsub::subject<T>(cs));
            static_assert(sizeof...(AN) == 0, "publish() was passed too many arguments.");
        }

        /*! Turn a cold observable hot and allow connections to the source to be independent of subscriptions.

        \param  cs  the subscription to control lifetime

        \return  rxcpp::connectable_observable that upon connection causes the source observable to emit items to its observers.

        \sample
        \snippet publish.cpp publish subject sample
        \snippet output.txt publish subject sample
        */
        template<class... AN>
        auto publish(composite_subscription cs, AN**...) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS multicast(rxsub::subject<T>(cs)))
            /// \endcond
        {
            return          multicast(rxsub::subject<T>(cs));
            static_assert(sizeof...(AN) == 0, "publish(composite_subscription) was passed too many arguments.");
        }

        /*! Turn a cold observable hot, send the most recent value to any new subscriber, and allow connections to the source to be
independent of subscriptions.

        \tparam  T  the type of the emitted item

        \param  first  an initial item to be emitted by the resulting observable at connection time before emitting the items from the
source observable; not emitted to observers that subscribe after the time of connection

        \return  rxcpp::connectable_observable that upon connection causes the source observable to emit items to its observers.

        \sample
        \snippet publish.cpp publish behavior sample
        \snippet output.txt publish behavior sample
        */
        template<class... AN>
        auto publish(T first, AN**...) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS multicast(rxsub::behavior<T>(first, composite_subscription())))
            /// \endcond
        {
            composite_subscription cs;
            return      multicast(rxsub::behavior<T>(first, cs));
            static_assert(sizeof...(AN) == 0, "publish(value_type) was passed too many arguments.");
        }

        /*! Turn a cold observable hot, send the most recent value to any new subscriber, and allow connections to the source to be
independent of subscriptions.

        \tparam  T  the type of the emitted item

        \param  first  an initial item to be emitted by the resulting observable at connection time before emitting the items from the
source observable; not emitted to observers that subscribe after the time of connection
        \param  cs     the subscription to control lifetime

        \return  rxcpp::connectable_observable that upon connection causes the source observable to emit items to its observers.

        \sample
        \snippet publish.cpp publish behavior sample
        \snippet output.txt publish behavior sample
        */
        template<class... AN>
        auto publish(T first, composite_subscription cs, AN**...) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS multicast(rxsub::behavior<T>(first, cs)))
            /// \endcond
        {
            return      multicast(rxsub::behavior<T>(first, cs));
            static_assert(sizeof...(AN) == 0, "publish(value_type, composite_subscription) was passed too many
arguments.");
        }

        /*! Turn a cold observable hot, send all earlier emitted values to any new subscriber, and allow connections to the source to
be independent of subscriptions.

        \return  rxcpp::connectable_observable that shares a single subscription to the underlying observable that will replay all of
its items and notifications to any future observer.
```

```
                \sample
                \snippet replay.cpp replay sample
                \snippet output.txt replay sample
                */
                template<class... AN>
                auto replay(AN**...) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS multicast(rxsub::replay<T, identity_one_worker>(identity_current_thread(),
composite_subscription())))
                        /// \endcond
                {
                        composite_subscription cs;
                        return      multicast(rxsub::replay<T, identity_one_worker>(identity_current_thread(), cs));
                        static_assert(sizeof...(AN) == 0, "replay() was passed too many arguments.");
                }

                /*! Turn a cold observable hot, send all earlier emitted values to any new subscriber, and allow connections to the source to
be independent of subscriptions.

                \param  cs  the subscription to control lifetime

                \return  rxcpp::connectable_observable that shares a single subscription to the underlying observable that will replay all of
its items and notifications to any future observer.

                \sample
                \snippet replay.cpp replay sample
                \snippet output.txt replay sample
                */
                template<class... AN>
                auto replay(composite_subscription cs, AN**...) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS multicast(rxsub::replay<T, identity_one_worker>(identity_current_thread(), cs)))
                        /// \endcond
                {
                        return      multicast(rxsub::replay<T, identity_one_worker>(identity_current_thread(), cs));
                        static_assert(sizeof...(AN) == 0, "replay(composite_subscription) was passed too many arguments.");
                }

                /*! Turn a cold observable hot, send all earlier emitted values to any new subscriber, and allow connections to the source to
be independent of subscriptions.

                \tparam  Coordination  the type of the scheduler

                \param  cn  a scheduler all values are queued and delivered on
                \param  cs  the subscription to control lifetime

                \return  rxcpp::connectable_observable that shares a single subscription to the underlying observable that will replay all of
its items and notifications to any future observer.

                \sample
                \snippet replay.cpp threaded replay sample
                \snippet output.txt threaded replay sample
                */
                template<class Coordination,
                class Requires = typename std::enable_if<is_coordination<Coordination>::value, rxu::types_checked>::type>
                        auto replay(Coordination cn, composite_subscription cs = composite_subscription()) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS multicast(rxsub::replay<T, Coordination>(std::move(cn), cs)))
                        /// \endcond
                {
                                return      multicast(rxsub::replay<T, Coordination>(std::move(cn), cs));
                        }

                /*! Turn a cold observable hot, send at most count of earlier emitted values to any new subscriber, and allow connections to
the source to be independent of subscriptions.

                \param  count  the maximum number of the most recent items sent to new observers

                \return  rxcpp::connectable_observable that shares a single subscription to the underlying observable that will replay at most
count items to any future observer.

                \sample
                \snippet replay.cpp replay count sample
                \snippet output.txt replay count sample
                */
                template<class... AN>
                auto replay(std::size_t count, AN**...) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS multicast(rxsub::replay<T, identity_one_worker>(count, identity_current_thread(),
composite_subscription())))
                        /// \endcond
```

```
                {
                        composite_subscription cs;
                        return      multicast(rxsub::replay<T, identity_one_worker>(count, identity_current_thread(), cs));
                        static_assert(sizeof...(AN) == 0, "replay(count) was passed too many arguments.");
                }

                /*! Turn a cold observable hot, send at most count of earlier emitted values to any new subscriber, and allow connections to
the source to be independent of subscriptions.

                \param  count  the maximum number of the most recent items sent to new observers
                \param  cs     the subscription to control lifetime

                \return  rxcpp::connectable_observable that shares a single subscription to the underlying observable that will replay at most
count items to any future observer.

                \sample
                \snippet replay.cpp replay count sample
                \snippet output.txt replay count sample
                */
                template<class... AN>
                auto replay(std::size_t count, composite_subscription cs, AN**...) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS multicast(rxsub::replay<T, identity_one_worker>(count, identity_current_thread(),
cs)))
                        /// \endcond
                {
                        return      multicast(rxsub::replay<T, identity_one_worker>(count, identity_current_thread(), cs));
                        static_assert(sizeof...(AN) == 0, "replay(count, composite_subscription) was passed too many arguments.");
                }

                /*! Turn a cold observable hot, send at most count of earlier emitted values to any new subscriber, and allow connections to
the source to be independent of subscriptions.

                \tparam  Coordination  the type of the scheduler

                \param  count  the maximum number of the most recent items sent to new observers
                \param  cn     a scheduler all values are queued and delivered on
                \param  cs     the subscription to control lifetime

                \return  rxcpp::connectable_observable that shares a single subscription to the underlying observable that will replay at most
count items to any future observer.

                \sample
                \snippet replay.cpp threaded replay count sample
                \snippet output.txt threaded replay count sample
                */
                template<class Coordination,
                class Requires = typename std::enable_if<is_coordination<Coordination>::value, rxu::types_checked>::type>
                        auto replay(std::size_t count, Coordination cn, composite_subscription cs = composite_subscription()) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS multicast(rxsub::replay<T, Coordination>(count, std::move(cn), cs)))
                        /// \endcond
                {
                                return      multicast(rxsub::replay<T, Coordination>(count, std::move(cn), cs));
                        }

                /*! Turn a cold observable hot, send values emitted within a specified time window to any new subscriber, and allow
connections to the source to be independent of subscriptions.

                \param  period  the duration of the window in which the replayed items must be emitted
                \param  cs      the subscription to control lifetime

                \return  rxcpp::connectable_observable that shares a single subscription to the underlying observable that will replay items
emitted within a specified time window to any future observer.

                \sample
                \snippet replay.cpp replay period sample
                \snippet output.txt replay period sample
                */
                template<class Duration>
                auto replay(Duration period, composite_subscription cs = composite_subscription()) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS multicast(rxsub::replay<T, identity_one_worker>(period,
identity_current_thread(), cs)))
                        /// \endcond
                {
                        return      multicast(rxsub::replay<T, identity_one_worker>(period, identity_current_thread(), cs));
                }

                /*! Turn a cold observable hot, send values emitted within a specified time window to any new subscriber, and allow
connections to the source to be independent of subscriptions.
```

```
                \tparam  Coordination  the type of the scheduler

                \param  period  the duration of the window in which the replayed items must be emitted
                \param  cn      a scheduler all values are queued and delivered on
                \param  cs      the subscription to control lifetime

                \return  rxcpp::connectable_observable that shares a single subscription to the underlying observable that will replay items emitted within a specified time window to any future observer.

                \sample
                \snippet replay.cpp threaded replay period sample
                \snippet output.txt threaded replay period sample
                */
                template<class Coordination,
                class Requires = typename std::enable_if<is_coordination<Coordination>::value, rxu::types_checked>::type>
                        auto replay(rxsc::scheduler::clock_type::duration period, Coordination cn, composite_subscription cs = composite_subscription()) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS multicast(rxsub::replay<T, Coordination>(period, std::move(cn), cs)))
                        /// \endcond
                {
                                return      multicast(rxsub::replay<T, Coordination>(period, std::move(cn), cs));
                        }

                /*! Turn a cold observable hot, send at most count of values emitted within a specified time window to any new subscriber, and allow connections to the source to be independent of subscriptions.

                \param  count   the maximum number of the most recent items sent to new observers
                \param  period  the duration of the window in which the replayed items must be emitted
                \param  cs      the subscription to control lifetime

                \return  rxcpp::connectable_observable that shares a single subscription to the underlying observable that will replay at most count of items emitted within a specified time window to any future observer.

                \sample
                \snippet replay.cpp replay count+period sample
                \snippet output.txt replay count+period sample
                */
                template<class Duration>
                auto replay(std::size_t count, Duration period, composite_subscription cs = composite_subscription()) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS multicast(rxsub::replay<T, identity_one_worker>(count, period, identity_current_thread(), cs)))
                        /// \endcond
                {
                        return      multicast(rxsub::replay<T, identity_one_worker>(count, period, identity_current_thread(), cs));
                }

                /*! Turn a cold observable hot, send at most count of values emitted within a specified time window to any new subscriber, and allow connections to the source to be independent of subscriptions.

                \tparam  Coordination  the type of the scheduler

                \param  count   the maximum number of the most recent items sent to new observers
                \param  period  the duration of the window in which the replayed items must be emitted
                \param  cn      a scheduler all values are queued and delivered on
                \param  cs      the subscription to control lifetime

                \return  rxcpp::connectable_observable that shares a single subscription to the underlying observable that will replay at most count of items emitted within a specified time window to any future observer.

                \sample
                \snippet replay.cpp threaded replay count+period sample
                \snippet output.txt threaded replay count+period sample
                */
                template<class Coordination,
                class Requires = typename std::enable_if<is_coordination<Coordination>::value, rxu::types_checked>::type>
                        auto replay(std::size_t count, rxsc::scheduler::clock_type::duration period, Coordination cn, composite_subscription cs = composite_subscription()) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(EXPLICIT_THIS multicast(rxsub::replay<T, Coordination>(count, period, std::move(cn), cs)))
                        /// \endcond
                {
                                return      multicast(rxsub::replay<T, Coordination>(count, period, std::move(cn), cs));
                        }

                /*! Subscription and unsubscription are queued and delivered using the scheduler from the supplied coordination.

                \tparam Coordination  the type of the scheduler

                \param  cn  the scheduler to perform subscription actions on
```

```
                        \return  The source observable modified so that its subscriptions happen on the specified scheduler.

                        \sample
                        \snippet subscribe_on.cpp subscribe_on sample
                        \snippet output.txt subscribe_on sample

                        Invoking rxcpp::observable::observe_on operator, instead of subscribe_on, gives following results:
                        \snippet output.txt observe_on sample
                        */
                        template<class Coordination>
                        auto subscribe_on(Coordination cn) const
                                /// \cond SHOW_SERVICE_MEMBERS
                                ->      observable<rxu::value_type_t<rxo::detail::subscribe_on<T, this_type, Coordination>>,
rxo::detail::subscribe_on<T, this_type, Coordination>>
                                /// \endcond
                        {
                                return  observable<rxu::value_type_t<rxo::detail::subscribe_on<T, this_type, Coordination>>,
rxo::detail::subscribe_on<T, this_type, Coordination>>(
                                        rxo::detail::subscribe_on<T, this_type, Coordination>(*this, std::move(cn)));
                        }

                        /*! All values are queued and delivered using the scheduler from the supplied coordination.

                        \tparam Coordination  the type of the scheduler

                        \param  cn  the scheduler to notify observers on

                        \return  The source observable modified so that its observers are notified on the specified scheduler.

                        \sample
                        \snippet observe_on.cpp observe_on sample
                        \snippet output.txt observe_on sample

                        Invoking rxcpp::observable::subscribe_on operator, instead of observe_on, gives following results:
                        \snippet output.txt subscribe_on sample
                        */
                        template<class Coordination>
                        auto observe_on(Coordination cn) const
                                /// \cond SHOW_SERVICE_MEMBERS
                                -> decltype(EXPLICIT_THIS lift<T>(rxo::detail::observe_on<T, Coordination>(std::move(cn))))
                                /// \endcond
                        {
                                return                  lift<T>(rxo::detail::observe_on<T, Coordination>(std::move(cn)));
                        }

                        /*! @copydoc rx-reduce.hpp
                        */
                        template<class... AN>
                        auto reduce(AN&&... an) const
                                /// \cond SHOW_SERVICE_MEMBERS
                                -> decltype(observable_member(reduce_tag{}, *(this_type*)nullptr, std::forward<AN>(an)...))
                                /// \endcond
                        {
                                return      observable_member(reduce_tag{}, *this, std::forward<AN>(an)...);
                        }

                        /*! @copydoc rxcpp::operators::first
                        */
                        template<class... AN>
                        auto first(AN**...) const
                                /// \cond SHOW_SERVICE_MEMBERS
                                -> decltype(observable_member(delayed_type<first_tag, AN...>::value(), *(this_type*)nullptr))
                                /// \endcond
                        {
                                return      observable_member(delayed_type<first_tag, AN...>::value(), *this);
                                static_assert(sizeof...(AN) == 0, "first() was passed too many arguments.");
                        }

                        /*! @copydoc rxcpp::operators::last
                        */
                        template<class... AN>
                        auto last(AN**...) const
                                /// \cond SHOW_SERVICE_MEMBERS
                                -> decltype(observable_member(delayed_type<last_tag, AN...>::value(), *(this_type*)nullptr))
                                /// \endcond
                        {
                                return      observable_member(delayed_type<last_tag, AN...>::value(), *this);
                                static_assert(sizeof...(AN) == 0, "last() was passed too many arguments.");
                        }

                        /*! @copydoc rxcpp::operators::count
                        */
```

```
                template<class... AN>
                auto count(AN**...) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(observable_member(delayed_type<reduce_tag, AN...>::value(), *(this_type*)nullptr, 0, rxu::count(),
identity_for<int>()))
                        /// \endcond
                {
                        return  observable_member(delayed_type<reduce_tag, AN...>::value(), *this, 0, rxu::count(),
identity_for<int>());
                        static_assert(sizeof...(AN) == 0, "count() was passed too many arguments.");
                }

                /*! @copydoc rxcpp::operators::sum
                */
                template<class... AN>
                auto sum(AN**...) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(observable_member(delayed_type<sum_tag, AN...>::value(), *(this_type*)nullptr))
                        /// \endcond
                {
                        return  observable_member(delayed_type<sum_tag, AN...>::value(), *this);
                        static_assert(sizeof...(AN) == 0, "sum() was passed too many arguments.");
                }

                /*! @copydoc rxcpp::operators::average
                */
                template<class... AN>
                auto average(AN**...) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(observable_member(delayed_type<average_tag, AN...>::value(), *(this_type*)nullptr))
                        /// \endcond
                {
                        return  observable_member(delayed_type<average_tag, AN...>::value(), *this);
                        static_assert(sizeof...(AN) == 0, "average() was passed too many arguments.");
                }

                /*! @copydoc rxcpp::operators::max
                */
                template<class... AN>
                auto max(AN**...) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(observable_member(delayed_type<max_tag, AN...>::value(), *(this_type*)nullptr))
                        /// \endcond
                {
                        return  observable_member(delayed_type<max_tag, AN...>::value(), *this);
                        static_assert(sizeof...(AN) == 0, "max() was passed too many arguments.");
                }

                /*! @copydoc rxcpp::operators::min
                */
                template<class... AN>
                auto min(AN**...) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> decltype(observable_member(delayed_type<min_tag, AN...>::value(), *(this_type*)nullptr))
                        /// \endcond
                {
                        return  observable_member(delayed_type<min_tag, AN...>::value(), *this);
                        static_assert(sizeof...(AN) == 0, "min() was passed too many arguments.");
                }

                /*! For each item from this observable use Accumulator to combine items into a value that will be emitted from the new
observable that is returned.

                \tparam Seed        the type of the initial value for the accumulator
                \tparam Accumulator  the type of the data accumulating function

                \param seed  the initial value for the accumulator
                \param a     an accumulator function to be invoked on each item emitted by the source observable, whose result will be
emitted and used in the next accumulator call

                \return  An observable that emits the results of each call to the accumulator function.

                \sample
                \snippet scan.cpp scan sample
                \snippet output.txt scan sample
                */
                template<class Seed, class Accumulator>
                auto scan(Seed seed, Accumulator&& a) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        ->      observable<Seed, rxo::detail::scan<T, this_type, Accumulator, Seed>>
                        /// \endcond
                {
```

```
                return  observable<Seed, rxo::detail::scan<T, this_type, Accumulator, Seed>>(
                        rxo::detail::scan<T, this_type, Accumulator, Seed>(*this, std::forward<Accumulator>(a), seed));
            }

            /*! Return an Observable that emits the most recent items emitted by the source Observable within periodic time intervals.

            \param period  the period of time to sample the source observable.
            \param coordination  the scheduler for the items.

            \return  Observable that emits the most recently emitted item since the previous sampling.

            \sample
            \snippet sample.cpp sample period sample
            \snippet output.txt sample period sample
            */
            template<class Coordination,
            class Requires = typename std::enable_if<is_coordination<Coordination>::value, rxu::types_checked>::type>
                auto sample_with_time(rxsc::scheduler::clock_type::duration period, Coordination coordination) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(EXPLICIT_THIS lift<T>(rxo::detail::sample_with_time<T, rxsc::scheduler::clock_type::duration,
Coordination>(period, coordination)))
                /// \endcond
            {
                    return      lift<T>(rxo::detail::sample_with_time<T, rxsc::scheduler::clock_type::duration,
Coordination>(period, coordination));
                }

            /*! Return an Observable that emits the most recent items emitted by the source Observable within periodic time intervals.

            \param period  the period of time to sample the source observable.

            \return  Observable that emits the most recently emitted item since the previous sampling.

            \sample
            \snippet sample.cpp sample period sample
            \snippet output.txt sample period sample
            */
            template<class... AN>
            auto sample_with_time(rxsc::scheduler::clock_type::duration period, AN**...) const
                /// \cond SHOW_SERVICE_MEMBERS
                -> decltype(EXPLICIT_THIS lift<T>(rxo::detail::sample_with_time<T, rxsc::scheduler::clock_type::duration,
identity_one_worker>(period, identity_current_thread())))
                /// \endcond
            {
                return      lift<T>(rxo::detail::sample_with_time<T, rxsc::scheduler::clock_type::duration,
identity_one_worker>(period, identity_current_thread()));
                static_assert(sizeof...(AN) == 0, "sample_with_time(period) was passed too many arguments.");
            }

            /*! Make new observable with skipped first count items from this observable.

            \tparam  Count  the type of the items counter

            \param  t  the number of items to skip

            \return  An observable that is identical to the source observable except that it does not emit the first t items that the source
observable emits.

            \sample
            \snippet skip.cpp skip sample
            \snippet output.txt skip sample
            */
            template<class Count>
            auto skip(Count t) const
                /// \cond SHOW_SERVICE_MEMBERS
                ->      observable<T, rxo::detail::skip<T, this_type, Count>>
                /// \endcond
            {
                return  observable<T, rxo::detail::skip<T, this_type, Count>>(
                        rxo::detail::skip<T, this_type, Count>(*this, t));
            }

            /*! Make new observable with skipped last count items from this observable.

            \tparam  Count  the type of the items counter

            \param  t  the number of last items to skip

            \return  An observable that is identical to the source observable except that it does not emit the last t items that the source
observable emits.

            \sample
```

```
                        \snippet skip_last.cpp skip_last sample
                        \snippet output.txt skip_last sample
                        */
                        template<class Count>
                        auto skip_last(Count t) const
                                /// \cond SHOW_SERVICE_MEMBERS
                                ->      observable<T, rxo::detail::skip_last<T, this_type, Count>>
                                /// \endcond
                        {
                                return  observable<T, rxo::detail::skip_last<T, this_type, Count>>(
                                        rxo::detail::skip_last<T, this_type, Count>(*this, t));
                        }

                        /*! Make new observable with items skipped until on_next occurs on the trigger observable

                        \tparam  TriggerSource  the type of the trigger observable

                        \param  t  an observable that has to emit an item before the source observable's elements begin to be mirrored by the
resulting observable

                        \return  An observable that skips items from the source observable until the second observable emits an item, then emits the
remaining items.

                        \note All sources must be synchronized! This means that calls across all the subscribers must be serial.

                        \sample
                        \snippet skip_until.cpp skip_until sample
                        \snippet output.txt skip_until sample
                        */
                        template<class TriggerSource>
                        auto skip_until(TriggerSource&& t) const
                                /// \cond SHOW_SERVICE_MEMBERS
                                -> typename std::enable_if<is_observable<TriggerSource>::value,
                                observable<T, rxo::detail::skip_until<T, this_type, TriggerSource, identity_one_worker>>>::type
                                /// \endcond
                        {
                                return  observable<T, rxo::detail::skip_until<T, this_type, TriggerSource, identity_one_worker>>(
                                        rxo::detail::skip_until<T, this_type, TriggerSource, identity_one_worker>(*this,
std::forward<TriggerSource>(t), identity_one_worker(rxsc::make_current_thread())));
                        }

                        /*! Make new observable with items skipped until on_next occurs on the trigger observable

                        \tparam  TriggerSource  the type of the trigger observable
                        \tparam  Coordination   the type of the scheduler

                        \param  t   an observable that has to emit an item before the source observable's elements begin to be mirrored by the
resulting observable
                        \param  cn  the scheduler to use for scheduling the items

                        \return  An observable that skips items from the source observable until the second observable emits an item, then emits the
remaining items.

                        \sample
                        \snippet skip_until.cpp threaded skip_until sample
                        \snippet output.txt threaded skip_until sample
                        */
                        template<class TriggerSource, class Coordination>
                        auto skip_until(TriggerSource&& t, Coordination&& cn) const
                                /// \cond SHOW_SERVICE_MEMBERS
                                -> typename std::enable_if<is_observable<TriggerSource>::value && is_coordination<Coordination>::value,
                                observable<T, rxo::detail::skip_until<T, this_type, TriggerSource, Coordination>>>::type
                                /// \endcond
                        {
                                return  observable<T, rxo::detail::skip_until<T, this_type, TriggerSource, Coordination>>(
                                        rxo::detail::skip_until<T, this_type, TriggerSource, Coordination>(*this,
std::forward<TriggerSource>(t), std::forward<Coordination>(cn)));
                        }

                        /*! For the first count items from this observable emit them from the new observable that is returned.

                        \tparam Count  the type of the items counter

                        \param t  the number of items to take

                        \return  An observable that emits only the first t items emitted by the source Observable, or all of the items from the source
observable if that observable emits fewer than t items.

                        \sample
                        \snippet take.cpp take sample
                        \snippet output.txt take sample
                        */
```

```
        template<class Count>
        auto take(Count t) const
            /// \cond SHOW_SERVICE_MEMBERS
            ->      observable<T, rxo::detail::take<T, this_type, Count>>
            /// \endcond
        {
            return  observable<T, rxo::detail::take<T, this_type, Count>>(
                        rxo::detail::take<T, this_type, Count>(*this, t));
        }

        /*! Emit only the final t items emitted by the source Observable.

        \tparam Count  the type of the items counter

        \param t  the number of last items to take

        \return  An observable that emits only the last t items emitted by the source Observable, or all of the items from the source observable if that observable emits fewer than t items.

        \sample
        \snippet take_last.cpp take_last sample
        \snippet output.txt take_last sample
        */
        template<class Count>
        auto take_last(Count t) const
            /// \cond SHOW_SERVICE_MEMBERS
            ->      observable<T, rxo::detail::take_last<T, this_type, Count>>
            /// \endcond
        {
            return  observable<T, rxo::detail::take_last<T, this_type, Count>>(
                        rxo::detail::take_last<T, this_type, Count>(*this, t));
        }


        /*! For each item from this observable until on_next occurs on the trigger observable, emit them from the new observable that is returned.

        \tparam  TriggerSource  the type of the trigger observable

        \param  t  an observable whose first emitted item will stop emitting items from the source observable

        \return  An observable that emits the items emitted by the source observable until such time as other emits its first item.

        \note All sources must be synchronized! This means that calls across all the subscribers must be serial.

        \sample
        \snippet take_until.cpp take_until sample
        \snippet output.txt take_until sample
        */
        template<class TriggerSource>
        auto take_until(TriggerSource t) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> typename std::enable_if<is_observable<TriggerSource>::value,
            observable<T, rxo::detail::take_until<T, this_type, TriggerSource, identity_one_worker>>>::type
            /// \endcond
        {
            return  observable<T, rxo::detail::take_until<T, this_type, TriggerSource, identity_one_worker>>(
                        rxo::detail::take_until<T, this_type, TriggerSource, identity_one_worker>(*this, std::move(t), identity_current_thread()));
        }

        /*! For each item from this observable until on_next occurs on the trigger observable, emit them from the new observable that is returned.

        \tparam  TriggerSource  the type of the trigger observable
        \tparam  Coordination   the type of the scheduler

        \param  t   an observable whose first emitted item will stop emitting items from the source observable
        \param  cn  the scheduler to use for scheduling the items

        \return  An observable that emits the items emitted by the source observable until such time as other emits its first item.

        \sample
        \snippet take_until.cpp threaded take_until sample
        \snippet output.txt threaded take_until sample
        */
        template<class TriggerSource, class Coordination>
        auto take_until(TriggerSource t, Coordination cn) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> typename std::enable_if<is_observable<TriggerSource>::value && is_coordination<Coordination>::value,
            observable<T, rxo::detail::take_until<T, this_type, TriggerSource, Coordination>>>::type
            /// \endcond
```

```
                {
                        return  observable<T, rxo::detail::take_until<T, this_type, TriggerSource, Coordination>>(
                                rxo::detail::take_until<T, this_type, TriggerSource, Coordination>(*this, std::move(t),
std::move(cn)));
                }

                /*! For each item from this observable until the specified time, emit them from the new observable that is returned.

                \tparam  TimePoint  the type of the time interval

                \param  when  an observable whose first emitted item will stop emitting items from the source observable

                \return  An observable that emits those items emitted by the source observable before the time runs out.

                \note All sources must be synchronized! This means that calls across all the subscribers must be serial.

                \sample
                \snippet take_until.cpp take_until time sample
                \snippet output.txt take_until time sample
                */
                template<class TimePoint>
                auto take_until(TimePoint when) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> typename std::enable_if<std::is_convertible<TimePoint, rxsc::scheduler::clock_type::time_point>::value,
                        observable<T, rxo::detail::take_until<T, this_type, decltype(rxs::timer(when, identity_current_thread())),
identity_one_worker>>>::type
                        /// \endcond
                {
                        auto cn = identity_current_thread();
                        return  take_until(rxs::timer(when, cn), cn);
                }

                /*! For each item from this observable until the specified time, emit them from the new observable that is returned.

                \tparam  TimePoint     the type of the time interval
                \tparam  Coordination  the type of the scheduler

                \param  when  an observable whose first emitted item will stop emitting items from the source observable
                \param  cn    the scheduler to use for scheduling the items

                \return  An observable that emits those items emitted by the source observable before the time runs out.

                \sample
                \snippet take_until.cpp threaded take_until time sample
                \snippet output.txt threaded take_until time sample
                */
                template<class Coordination>
                auto take_until(rxsc::scheduler::clock_type::time_point when, Coordination cn) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        -> typename std::enable_if<is_coordination<Coordination>::value,
                        observable<T, rxo::detail::take_until<T, this_type, decltype(rxs::timer(when, cn)), Coordination>>>::type
                        /// \endcond
                {
                        return  take_until(rxs::timer(when, cn), cn);
                }

                /*! Infinitely repeat this observable.

                \return  An observable that emits the items emitted by the source observable repeatedly and in sequence.

                \sample
                \snippet repeat.cpp repeat sample
                \snippet output.txt repeat sample

                If the source observable calls on_error, repeat stops:
                \snippet repeat.cpp repeat error sample
                \snippet output.txt repeat error sample
                */
                template<class... AN>
                auto repeat(AN**...) const
                        /// \cond SHOW_SERVICE_MEMBERS
                        ->      observable<T, rxo::detail::repeat<T, this_type, int>>
                        /// \endcond
                {
                        return  observable<T, rxo::detail::repeat<T, this_type, int>>(
                                rxo::detail::repeat<T, this_type, int>(*this, 0));
                        static_assert(sizeof...(AN) == 0, "repeat() was passed too many arguments.");
                }

                /*! Repeat this observable for the given number of times.

                \tparam Count  the type of the counter
```

```
\param t  the number of times the source observable items are repeated

\return  An observable that repeats the sequence of items emitted by the source observable for t times.

Call to repeat(0) infinitely repeats the source observable.

\sample
\snippet repeat.cpp repeat count sample
\snippet output.txt repeat count sample
*/
template<class Count>
auto repeat(Count t) const
    /// \cond SHOW_SERVICE_MEMBERS
    ->      observable<T, rxo::detail::repeat<T, this_type, Count>>
    /// \endcond
{
    return  observable<T, rxo::detail::repeat<T, this_type, Count>>(
            rxo::detail::repeat<T, this_type, Count>(*this, t));
}

/*! Infinitely retry this observable.

\return  An observable that mirrors the source observable, resubscribing to it if it calls on_error.

\sample
\snippet retry.cpp retry sample
\snippet output.txt retry sample
*/
template<class... AN>
auto retry(AN**...) const
    /// \cond SHOW_SERVICE_MEMBERS
    ->      observable<T, rxo::detail::retry<T, this_type, int>>
    /// \endcond
{
    return  observable<T, rxo::detail::retry<T, this_type, int>>(
            rxo::detail::retry<T, this_type, int>(*this, 0));
    static_assert(sizeof...(AN) == 0, "retry() was passed too many arguments.");
}

/*! Retry this observable for the given number of times.

\tparam Count  the type of the counter

\param t  the number of retries

\return  An observable that mirrors the source observable, resubscribing to it if it calls on_error up to a specified number of
retries.

Call to retry(0) infinitely retries the source observable.

\sample
\snippet retry.cpp retry count sample
\snippet output.txt retry count sample
*/
template<class Count>
auto retry(Count t) const
    /// \cond SHOW_SERVICE_MEMBERS
    ->      observable<T, rxo::detail::retry<T, this_type, Count>>
    /// \endcond
{
    return  observable<T, rxo::detail::retry<T, this_type, Count>>(
            rxo::detail::retry<T, this_type, Count>(*this, t));
}

/*! Start with the supplied values, then concatenate this observable.

\tparam Value0      ...
\tparam ValueN      the type of sending values

\param  v0  ...
\param  vn  values to send

\return  Observable that emits the specified items and then emits the items emitted by the source observable.

\sample
\snippet start_with.cpp short start_with sample
\snippet output.txt short start_with sample

Another form of this operator, rxcpp::observable<void, void>::start_with, gets the source observable as a parameter:
\snippet start_with.cpp full start_with sample
\snippet output.txt full start_with sample
```

```
        */
        template<class Value0, class... ValueN>
        auto start_with(Value0 v0, ValueN... vn) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(rxo::start_with(std::move(v0), std::move(vn)...)(*(this_type*)nullptr))
            /// \endcond
        {
            return      rxo::start_with(std::move(v0), std::move(vn)...)(*this);
        }

        /*! Take values pairwise from this observable.

        \return  Observable that emits tuples of two the most recent items emitted by the source observable.

        \sample
        \snippet pairwise.cpp pairwise sample
        \snippet output.txt pairwise sample

        If the source observable emits less than two items, no pairs are emitted  by the source observable:
        \snippet pairwise.cpp pairwise short sample
        \snippet output.txt pairwise short sample
        */
        template<class... AN>
        auto pairwise(AN**...) const
            /// \cond SHOW_SERVICE_MEMBERS
            -> decltype(EXPLICIT_THIS lift<rxu::value_type_t<rxo::detail::pairwise<T>>>(rxo::detail::pairwise<T>()))
            /// \endcond
        {
            return                  lift<rxu::value_type_t<rxo::detail::pairwise<T>>>(rxo::detail::pairwise<T>());
            static_assert(sizeof...(AN) == 0, "pairwise() was passed too many arguments.");
        }
};

template<class T, class SourceOperator>
inline bool operator==(const observable<T, SourceOperator>& lhs, const observable<T, SourceOperator>& rhs) {
    return lhs.source_operator == rhs.source_operator;
}
template<class T, class SourceOperator>
inline bool operator!=(const observable<T, SourceOperator>& lhs, const observable<T, SourceOperator>& rhs) {
    return !(lhs == rhs);
}

/*!
\defgroup group-core Basics

\brief These are the core classes that combine to represent a set of values emitted over time that can be cancelled.

\class rxcpp::observable<void, void>

\brief typed as ```rxcpp::observable<>```, this is a collection of factory methods that return an observable.

\ingroup group-core

\par Create a new type of observable

\sample
\snippet create.cpp Create sample
\snippet output.txt Create sample

\par Create an observable that emits a range of values

\sample
\snippet range.cpp range sample
\snippet output.txt range sample

\par Create an observable that emits nothing / generates an error / immediately completes

\sample
\snippet never.cpp never sample
\snippet output.txt never sample
\snippet error.cpp error sample
\snippet output.txt error sample
\snippet empty.cpp empty sample
\snippet output.txt empty sample

\par Create an observable that generates new observable for each subscriber

\sample
\snippet defer.cpp defer sample
\snippet output.txt defer sample

\par Create an observable that emits items every specified interval of time
```

```
        \sample
        \snippet interval.cpp interval sample
        \snippet output.txt interval sample

        \par Create an observable that emits items in the specified interval of time

        \sample
        \snippet timer.cpp duration timer sample
        \snippet output.txt duration timer sample

        \par Create an observable that emits all items from a collection

        \sample
        \snippet iterate.cpp iterate sample
        \snippet output.txt iterate sample

        \par Create an observable that emits a set of specified items

        \sample
        \snippet from.cpp from sample
        \snippet output.txt from sample

        \par Create an observable that emits a single item

        \sample
        \snippet just.cpp just sample
        \snippet output.txt just sample

        \par Create an observable that emits a set of items and then subscribes to another observable

        \sample
        \snippet start_with.cpp full start_with sample
        \snippet output.txt full start_with sample

        \par Create an observable that generates a new observable based on a generated resource for each subscriber

        \sample
        \snippet scope.cpp scope sample
        \snippet output.txt scope sample

        */
        template<>
        class observable<void, void>
        {
                ~observable();
        public:
                /*! Returns an observable that executes the specified function when a subscriber subscribes to it.

                \tparam T  the type of the items that this observable emits
                \tparam OnSubscribe  the type of OnSubscribe handler function

                \param  os  OnSubscribe event handler

                \return  Observable that executes the specified function when a Subscriber subscribes to it.

                \sample
                \snippet create.cpp Create sample
                \snippet output.txt Create sample

                \warning
                It is good practice to check the observer's is_subscribed state from within the function you pass to create
                so that your observable can stop emitting items or doing expensive calculations when there is no longer an interested
observer.

                \badcode
                \snippet create.cpp Create bad code
                \snippet output.txt Create bad code

                \goodcode
                \snippet create.cpp Create good code
                \snippet output.txt Create good code

                \warning
                It is good practice to use operators like observable::take to control lifetime rather than use the subscription explicitly.

                \goodcode
                \snippet create.cpp Create great code
                \snippet output.txt Create great code
                */
                template<class T, class OnSubscribe>
                static auto create(OnSubscribe os)
```

```
                -> decltype(rxs::create<T>(std::move(os))) {
                return      rxs::create<T>(std::move(os));
            }
            /*! Returns an observable that sends values in the range first-last by adding step to the previous value.

            \tparam T  the type of the values that this observable emits

            \param  first  first value to send
            \param  last   last value to send
            \param  step   value to add to the previous value to get the next value

            \return  Observable that sends values in the range first-last by adding step to the previous value.

            \sample
            \snippet range.cpp range sample
            \snippet output.txt range sample
            */
            template<class T>
            static auto range(T first = 0, T last = std::numeric_limits<T>::max(), std::ptrdiff_t step = 1)
                -> decltype(rxs::range<T>(first, last, step, identity_current_thread())) {
                return      rxs::range<T>(first, last, step, identity_current_thread());
            }
            /*! Returns an observable that sends values in the range ```first```-```last``` by adding ```step``` to the previous value. The values are sent on the specified scheduler.

            \tparam T            the type of the values that this observable emits
            \tparam Coordination  the type of the scheduler

            \param  first  first value to send
            \param  last   last value to send
            \param  step   value to add to the previous value to get the next value
            \param  cn     the scheduler to run the generator loop on

            \return  Observable that sends values in the range first-last by adding step to the previous value using the specified scheduler.

            \note  `step` or both `step` & `last` may be omitted.

            \sample
            \snippet range.cpp threaded range sample
            \snippet output.txt threaded range sample

            An alternative way to specify the scheduler for emitted values is to use observable::subscribe_on operator
            \snippet range.cpp subscribe_on range sample
            \snippet output.txt subscribe_on range sample
            */
            template<class T, class Coordination>
            static auto range(T first, T last, std::ptrdiff_t step, Coordination cn)
                -> decltype(rxs::range<T>(first, last, step, std::move(cn))) {
                return      rxs::range<T>(first, last, step, std::move(cn));
            }
            /// Returns an observable that sends values in the range ```first```-```last``` by adding 1 to the previous value. The values are sent on the specified scheduler.
            ///
            /// \see      rxcpp::observable<void,void>#range(T first, T last, std::ptrdiff_t step, Coordination cn)
            template<class T, class Coordination>
            static auto range(T first, T last, Coordination cn)
                -> decltype(rxs::range<T>(first, last, std::move(cn))) {
                return      rxs::range<T>(first, last, std::move(cn));
            }
            /// Returns an observable that infinitely (until overflow) sends values starting from ```first```. The values are sent on the specified scheduler.
            ///
            /// \see      rxcpp::observable<void,void>#range(T first, T last, std::ptrdiff_t step, Coordination cn)
            template<class T, class Coordination>
            static auto range(T first, Coordination cn)
                -> decltype(rxs::range<T>(first, std::move(cn))) {
                return      rxs::range<T>(first, std::move(cn));
            }
            /*! Returns an observable that never sends any items or notifications to observer.

            \tparam T  the type of (not) emitted items

            \return  Observable that never sends any items or notifications to observer.

            \sample
            \snippet never.cpp never sample
            \snippet output.txt never sample
            */
            template<class T>
            static auto never()
                -> decltype(rxs::never<T>()) {
```

```
                return      rxs::never<T>();
        }
        /*! Returns an observable that calls the specified observable factory to create an observable for each new observer that
subscribes.

        \tparam ObservableFactory  the type of the observable factory

        \param  of  the observable factory function to invoke for each observer that subscribes to the resulting observable

        \return  observable whose observers' subscriptions trigger an invocation of the given observable factory function

        \sample
        \snippet defer.cpp defer sample
        \snippet output.txt defer sample
        */
        template<class ObservableFactory>
        static auto defer(ObservableFactory of)
                -> decltype(rxs::defer(std::move(of))) {
                return      rxs::defer(std::move(of));
        }
        /*! Returns an observable that emits a sequential integer every specified time interval.

        \param  period   period between emitted values

        \return  Observable that sends a sequential integer each time interval

        \sample
        \snippet interval.cpp immediate interval sample
        \snippet output.txt immediate interval sample
        */
        template<class... AN>
        static auto interval(rxsc::scheduler::clock_type::duration period, AN**...)
                -> decltype(rxs::interval(period)) {
                return      rxs::interval(period);
                static_assert(sizeof...(AN) == 0, "interval(period) was passed too many arguments.");
        }
        /*! Returns an observable that emits a sequential integer every specified time interval, on the specified scheduler.

        \tparam Coordination  the type of the scheduler

        \param  period   period between emitted values
        \param  cn       the scheduler to use for scheduling the items

        \return  Observable that sends a sequential integer each time interval

        \sample
        \snippet interval.cpp threaded immediate interval sample
        \snippet output.txt threaded immediate interval sample
        */
        template<class Coordination>
        static auto interval(rxsc::scheduler::clock_type::duration period, Coordination cn)
                -> decltype(rxs::interval(period, std::move(cn))) {
                return      rxs::interval(period, std::move(cn));
        }
        /*! Returns an observable that emits a sequential integer every specified time interval starting from the specified time point.

        \param  initial  time when the first value is sent
        \param  period   period between emitted values

        \return  Observable that sends a sequential integer each time interval

        \sample
        \snippet interval.cpp interval sample
        \snippet output.txt interval sample
        */
        template<class... AN>
        static auto interval(rxsc::scheduler::clock_type::time_point initial, rxsc::scheduler::clock_type::duration period, AN**...)
                -> decltype(rxs::interval(initial, period)) {
                return      rxs::interval(initial, period);
                static_assert(sizeof...(AN) == 0, "interval(initial, period) was passed too many arguments.");
        }
        /*! Returns an observable that emits a sequential integer every specified time interval starting from the specified time point,
on the specified scheduler.

        \tparam Coordination  the type of the scheduler

        \param  initial  time when the first value is sent
        \param  period   period between emitted values
        \param  cn       the scheduler to use for scheduling the items

        \return  Observable that sends a sequential integer each time interval
```

```
\sample
\snippet interval.cpp threaded interval sample
\snippet output.txt threaded interval sample
*/
template<class Coordination>
static auto interval(rxsc::scheduler::clock_type::time_point initial, rxsc::scheduler::clock_type::duration period,
Coordination cn)
    -> decltype(rxs::interval(initial, period, std::move(cn))) {
    return      rxs::interval(initial, period, std::move(cn));
}
/*! Returns an observable that emits an integer at the specified time point.

\param  when  time point when the value is emitted

\return  Observable that emits an integer at the specified time point

\sample
\snippet timer.cpp timepoint timer sample
\snippet output.txt timepoint timer sample
*/
template<class... AN>
static auto timer(rxsc::scheduler::clock_type::time_point at, AN**...)
    -> decltype(rxs::timer(at)) {
    return      rxs::timer(at);
    static_assert(sizeof...(AN) == 0, "timer(at) was passed too many arguments.");
}
/*! Returns an observable that emits an integer in the specified time interval.

\param  when  interval when the value is emitted

\return  Observable that emits an integer in the specified time interval

\sample
\snippet timer.cpp duration timer sample
\snippet output.txt duration timer sample
*/
template<class... AN>
static auto timer(rxsc::scheduler::clock_type::duration after, AN**...)
    -> decltype(rxs::timer(after)) {
    return      rxs::timer(after);
    static_assert(sizeof...(AN) == 0, "timer(after) was passed too many arguments.");
}
/*! Returns an observable that emits an integer at the specified time point, on the specified scheduler.

\tparam Coordination  the type of the scheduler

\param  when  time point when the value is emitted
\param  cn    the scheduler to use for scheduling the items

\return  Observable that emits an integer at the specified time point

\sample
\snippet timer.cpp threaded timepoint timer sample
\snippet output.txt threaded timepoint timer sample
*/
template<class Coordination>
static auto timer(rxsc::scheduler::clock_type::time_point when, Coordination cn)
    -> decltype(rxs::timer(when, std::move(cn))) {
    return      rxs::timer(when, std::move(cn));
}
/*! Returns an observable that emits an integer in the specified time interval, on the specified scheduler.

\tparam Coordination  the type of the scheduler

\param  when  interval when the value is emitted
\param  cn    the scheduler to use for scheduling the items

\return  Observable that emits an integer in the specified time interval

\sample
\snippet timer.cpp threaded duration timer sample
\snippet output.txt threaded duration timer sample
*/
template<class Coordination>
static auto timer(rxsc::scheduler::clock_type::duration when, Coordination cn)
    -> decltype(rxs::timer(when, std::move(cn))) {
    return      rxs::timer(when, std::move(cn));
}
/*! Returns an observable that sends each value in the collection.

\tparam Collection  the type of the collection of values that this observable emits
```

```
                    \param  c  collection containing values to send

                    \return  Observable that sends each value in the collection.

                    \sample
                    \snippet iterate.cpp iterate sample
                    \snippet output.txt iterate sample
                    */
                    template<class Collection>
                    static auto iterate(Collection c)
                                -> decltype(rxs::iterate(std::move(c), identity_current_thread())) {
                                return      rxs::iterate(std::move(c), identity_current_thread());
                    }
                    /*! Returns an observable that sends each value in the collection, on the specified scheduler.

                    \tparam Collection    the type of the collection of values that this observable emits
                    \tparam Coordination  the type of the scheduler

                    \param  c   collection containing values to send
                    \param  cn  the scheduler to use for scheduling the items

                    \return  Observable that sends each value in the collection.

                    \sample
                    \snippet iterate.cpp threaded iterate sample
                    \snippet output.txt threaded iterate sample
                    */
                    template<class Collection, class Coordination>
                    static auto iterate(Collection c, Coordination cn)
                                -> decltype(rxs::iterate(std::move(c), std::move(cn))) {
                                return      rxs::iterate(std::move(c), std::move(cn));
                    }
                    /*! Returns an observable that sends an empty set of values and then completes.

                    \tparam T  the type of elements (not) to be sent

                    \return  Observable that sends an empty set of values and then completes.

                    This is a degenerate case of rxcpp::observable<void,void>#from(Value0,ValueN...) operator.

                    \note This is a degenerate case of ```observable<void,void>::from(Value0 v0, ValueN... vn)``` operator.
                    */
                    template<class T>
                    static auto from()
                                -> decltype(rxs::from<T>()) {
                                return          rxs::from<T>();
                    }
                    /*! Returns an observable that sends an empty set of values and then completes, on the specified scheduler.

                    \tparam T  the type of elements (not) to be sent
                    \tparam Coordination  the type of the scheduler

                    \return  Observable that sends an empty set of values and then completes.

                    \note This is a degenerate case of ```observable<void,void>::from(Coordination cn, Value0 v0, ValueN... vn)``` operator.
                    */
                    template<class T, class Coordination>
                    static auto from(Coordination cn)
                                -> typename std::enable_if<is_coordination<Coordination>::value,
                                decltype(rxs::from<T>(std::move(cn)))>::type {
                                return          rxs::from<T>(std::move(cn));
                    }
                    /*! Returns an observable that sends each value from its arguments list.

                    \tparam Value0  ...
                    \tparam ValueN  the type of sending values

                    \param  v0  ...
                    \param  vn  values to send

                    \return  Observable that sends each value from its arguments list.

                    \sample
                    \snippet from.cpp from sample
                    \snippet output.txt from sample

                    \note This operator is useful to send separated values. If they are stored as a collection, use observable<void,void>::iterate
instead.
                    */
                    template<class Value0, class... ValueN>
                    static auto from(Value0 v0, ValueN... vn)
                                -> typename std::enable_if<!is_coordination<Value0>::value,
```

```
                        decltype(rxs::from(v0, vn...))>::type {
                        return      rxs::from(v0, vn...);
                }
                /*! Returns an observable that sends each value from its arguments list, on the specified scheduler.

                \tparam Coordination  the type of the scheduler
                \tparam Value0  ...
                \tparam ValueN  the type of sending values

                \param  cn  the scheduler to use for scheduling the items
                \param  v0  ...
                \param  vn  values to send

                \return  Observable that sends each value from its arguments list.

                \sample
                \snippet from.cpp threaded from sample
                \snippet output.txt threaded from sample

                \note This operator is useful to send separated values. If they are stored as a collection, use observable<void,void>::iterate
instead.
                */
                template<class Coordination, class Value0, class... ValueN>
                static auto from(Coordination cn, Value0 v0, ValueN... vn)
                        -> typename std::enable_if<is_coordination<Coordination>::value,
                        decltype(rxs::from(std::move(cn), v0, vn...))>::type {
                        return      rxs::from(std::move(cn), v0, vn...);
                }
                /*! Returns an observable that sends no items to observer and immediately completes.

                \tparam T           the type of (not) emitted items

                \return  Observable that sends no items to observer and immediately completes.

                \sample
                \snippet empty.cpp empty sample
                \snippet output.txt empty sample
                */
                template<class T>
                static auto empty()
                        -> decltype(from<T>()) {
                        return      from<T>();
                }
                /*! Returns an observable that sends no items to observer and immediately completes, on the specified scheduler.

                \tparam T           the type of (not) emitted items
                \tparam Coordination  the type of the scheduler

                \param  cn  the scheduler to use for scheduling the items

                \return  Observable that sends no items to observer and immediately completes.

                \sample
                \snippet empty.cpp threaded empty sample
                \snippet output.txt threaded empty sample
                */
                template<class T, class Coordination>
                static auto empty(Coordination cn)
                        -> decltype(from<T>(std::move(cn))) {
                        return      from<T>(std::move(cn));
                }
                /*! Returns an observable that sends the specified item to observer and then completes.

                \tparam T  the type of the emitted item

                \param v  the value to send

                \return  Observable that sends the specified item to observer and then completes.

                \sample
                \snippet just.cpp just sample
                \snippet output.txt just sample
                */
                template<class T>
                static auto just(T v)
                        -> decltype(from(std::move(v))) {
                        return      from(std::move(v));
                }
                /*! Returns an observable that sends the specified item to observer and then completes, on the specified scheduler.

                \tparam T           the type of the emitted item
                \tparam Coordination  the type of the scheduler
```

```
\param v   the value to send
\param cn  the scheduler to use for scheduling the items

\return  Observable that sends the specified item to observer and then completes.

\sample
\snippet just.cpp threaded just sample
\snippet output.txt threaded just sample
*/
template<class T, class Coordination>
static auto just(T v, Coordination cn)
    -> decltype(from(std::move(cn), std::move(v))) {
    return      from(std::move(cn), std::move(v));
}
/*! Returns an observable that sends no items to observer and immediately generates an error.

\tparam T          the type of (not) emitted items
\tparam Exception  the type of the error

\param  e  the error to be passed to observers

\return  Observable that sends no items to observer and immediately generates an error.

\sample
\snippet error.cpp error sample
\snippet output.txt error sample
*/
template<class T, class Exception>
static auto error(Exception&& e)
    -> decltype(rxs::error<T>(std::forward<Exception>(e))) {
    return      rxs::error<T>(std::forward<Exception>(e));
}
/*! Returns an observable that sends no items to observer and immediately generates an error, on the specified scheduler.

\tparam T            the type of (not) emitted items
\tparam Exception     the type of the error
\tparam Coordination  the type of the scheduler

\param  e   the error to be passed to observers
\param  cn  the scheduler to use for scheduling the items

\return  Observable that sends no items to observer and immediately generates an error.

\sample
\snippet error.cpp threaded error sample
\snippet output.txt threaded error sample
*/
template<class T, class Exception, class Coordination>
static auto error(Exception&& e, Coordination cn)
    -> decltype(rxs::error<T>(std::forward<Exception>(e), std::move(cn))) {
    return      rxs::error<T>(std::forward<Exception>(e), std::move(cn));
}
/*! Returns an observable that sends the specified values before it begins to send items emitted by the given observable.

\tparam Observable  the type of the observable that emits values for resending
\tparam Value0      ...
\tparam ValueN      the type of sending values

\param  o   the observable that emits values for resending
\param  v0  ...
\param  vn  values to send

\return  Observable that sends the specified values before it begins to send items emitted by the given observable.

\sample
\snippet start_with.cpp full start_with sample
\snippet output.txt full start_with sample

    Instead of passing the observable as a parameter, you can use rxcpp::observable<T, SourceOperator>::start_with method of
the existing observable:
    \snippet start_with.cpp short start_with sample
    \snippet output.txt short start_with sample
    */
    template<class Observable, class Value0, class... ValueN>
    static auto start_with(Observable o, Value0 v0, ValueN... vn)
        -> decltype(rxs::from(rxu::value_type_t<Observable>(v0), rxu::value_type_t<Observable>(vn)...).concat(o)) {
        return      rxs::from(rxu::value_type_t<Observable>(v0), rxu::value_type_t<Observable>(vn)...).concat(o);
    }
    /*! Returns an observable that makes an observable by the specified observable factory
    using the resource provided by the specified resource factory for each new observer that subscribes.
```

```
                        \tparam ResourceFactory    the type of the resource factory
                        \tparam ObservableFactory  the type of the observable factory

                        \param  rf  the resource factory function that resturn the rxcpp::resource that is used as a resource by the observable factory
                        \param  of  the observable factory function to invoke for each observer that subscribes to the resulting observable

                        \return  observable that makes an observable by the specified observable factory
                        using the resource provided by the specified resource factory for each new observer that subscribes.

                        \sample
                        \snippet scope.cpp scope sample
                        \snippet output.txt scope sample
                        */
                        template<class ResourceFactory, class ObservableFactory>
                        static auto scope(ResourceFactory rf, ObservableFactory of)
                                -> decltype(rxs::scope(std::move(rf), std::move(of))) {
                                return      rxs::scope(std::move(rf), std::move(of));
                        }
        };

}

//
// support range() >> filter() >> subscribe() syntax
// '>>' is spelled 'stream'
//
template<class T, class SourceOperator, class OperatorFactory>
auto operator >> (const rxcpp::observable<T, SourceOperator>& source, OperatorFactory&& of)
-> decltype(source.op(std::forward<OperatorFactory>(of))) {
        return      source.op(std::forward<OperatorFactory>(of));
}

//
// support range() | filter() | subscribe() syntax
// '|' is spelled 'pipe'
//
template<class T, class SourceOperator, class OperatorFactory>
auto operator | (const rxcpp::observable<T, SourceOperator>& source, OperatorFactory&& of)
-> decltype(source.op(std::forward<OperatorFactory>(of))) {
        return      source.op(std::forward<OperatorFactory>(of));
}

#endif


#if !defined(RXCPP_RX_CONNECTABLE_OBSERVABLE_HPP)
#define RXCPP_RX_CONNECTABLE_OBSERVABLE_HPP

//_include "rx-includes.hpp"

namespace rxcpp {

        namespace detail {

                template<class T>
                struct has_on_connect
                {
                        struct not_void {};
                        template<class CT>
                        static auto check(int) -> decltype((*(CT*)nullptr).on_connect(composite_subscription()));
                        template<class CT>
                        static not_void check(...);

                        typedef decltype(check<T>(0)) detail_result;
                        static const bool value = std::is_same<detail_result, void>::value;
                };

        }

        template<class T>
        class dynamic_connectable_observable
                : public dynamic_observable<T>
        {
                struct state_type
                : public std::enable_shared_from_this<state_type>
                {
                        typedef std::function<void(composite_subscription)> onconnect_type;

                        onconnect_type on_connect;
                };
                std::shared_ptr<state_type> state;
```

```
                template<class U>
                void construct(const dynamic_observable<U>& o, tag_dynamic_observable&&) {
                        state = o.state;
                }

                template<class U>
                void construct(dynamic_observable<U>&& o, tag_dynamic_observable&&) {
                        state = std::move(o.state);
                }

                template<class SO>
                void construct(SO&& source, rxs::tag_source&&) {
                        auto so = std::make_shared<rxu::decay_t<SO>>(std::forward<SO>(source));
                        state->on_connect = [so](composite_subscription cs) mutable {
                                so->on_connect(std::move(cs));
                        };
                }

        public:

                typedef tag_dynamic_observable dynamic_observable_tag;

                dynamic_connectable_observable()
                {
                }

                template<class SOF>
                explicit dynamic_connectable_observable(SOF sof)
                        : dynamic_observable<T>(sof)
                        , state(std::make_shared<state_type>())
                {
                                construct(std::move(sof),
                                                typename std::conditional<is_dynamic_observable<SOF>::value,
tag_dynamic_observable, rxs::tag_source>::type());
                        }

                template<class SF, class CF>
                dynamic_connectable_observable(SF&& sf, CF&& cf)
                        : dynamic_observable<T>(std::forward<SF>(sf))
                        , state(std::make_shared<state_type>())
                {
                                state->on_connect = std::forward<CF>(cf);
                        }

                using dynamic_observable<T>::on_subscribe;

                void on_connect(composite_subscription cs) const {
                        state->on_connect(std::move(cs));
                }
        };

        template<class T, class Source>
        connectable_observable<T> make_dynamic_connectable_observable(Source&& s) {
                return connectable_observable<T>(dynamic_connectable_observable<T>(std::forward<Source>(s)));
        }


        /*!
        \brief a source of values that is shared across all subscribers and does not start until connectable_observable::connect() is called.

        \ingroup group-observable

        */
        template<class T, class SourceOperator>
        class connectable_observable
                : public observable<T, SourceOperator>
        {
                typedef connectable_observable<T, SourceOperator> this_type;
                typedef observable<T, SourceOperator> base_type;
                typedef rxu::decay_t<SourceOperator> source_operator_type;

                static_assert(detail::has_on_connect<source_operator_type>::value, "inner must have on_connect method
void(composite_subscription)");

        public:
                typedef tag_connectable_observable observable_tag;

                connectable_observable()
                {
                }

                explicit connectable_observable(const SourceOperator& o)
```

```
                : base_type(o)
        {
        }
        explicit connectable_observable(SourceOperator&& o)
                : base_type(std::move(o))
        {
        }

        // implicit conversion between observables of the same value_type
        template<class SO>
        connectable_observable(const connectable_observable<T, SO>& o)
                : base_type(o)
        {}
        // implicit conversion between observables of the same value_type
        template<class SO>
        connectable_observable(connectable_observable<T, SO>&& o)
                : base_type(std::move(o))
        {}

        ///
        /// takes any function that will take this observable and produce a result value.
        /// this is intended to allow externally defined operators, that use subscribe,
        /// to be connected into the expression.
        ///
        template<class OperatorFactory>
        auto op(OperatorFactory&& of) const
                -> decltype(of(*(const this_type*)nullptr)) {
                return  of(*this);
                static_assert(is_operator_factory_for<this_type, OperatorFactory>::value, "Function passed for op() must have
the signature Result(SourceObservable)");
        }

        ///
        /// performs type-forgetting conversion to a new composite_observable
        ///
        connectable_observable<T> as_dynamic() {
                return *this;
        }

        composite_subscription connect(composite_subscription cs = composite_subscription()) {
                base_type::source_operator.on_connect(cs);
                return cs;
        }

        /// ref_count ->
        /// takes a connectable_observable source and uses a ref_count of the subscribers
        /// to control the connection to the published source. The first subscription
        /// will cause a call to connect() and the last unsubscribe will unsubscribe the
        /// connection.
        ///
        auto ref_count() const
                ->  observable<T, rxo::detail::ref_count<T, this_type>> {
                return  observable<T, rxo::detail::ref_count<T, this_type>>(
                        rxo::detail::ref_count<T, this_type>(*this));
        }

        /// connect_forever ->
        /// takes a connectable_observable source and calls connect during
        /// the construction of the expression. This means that the source
        /// starts running without any subscribers and continues running
        /// after all subscriptions have been unsubscribed.
        ///
        auto connect_forever() const
                ->  observable<T, rxo::detail::connect_forever<T, this_type>> {
                return  observable<T, rxo::detail::connect_forever<T, this_type>>(
                        rxo::detail::connect_forever<T, this_type>(*this));
        }
    };


}

//
// support range() >> filter() >> subscribe() syntax
// '>>' is spelled 'stream'
//
template<class T, class SourceOperator, class OperatorFactory>
auto operator >> (const rxcpp::connectable_observable<T, SourceOperator>& source, OperatorFactory&& of)
-> decltype(source.op(std::forward<OperatorFactory>(of))) {
    return  source.op(std::forward<OperatorFactory>(of));
}
```

```
//
// support range() | filter() | subscribe() syntax
// '|' is spelled 'pipe'
//
template<class T, class SourceOperator, class OperatorFactory>
auto operator | (const rxcpp::connectable_observable<T, SourceOperator>& source, OperatorFactory&& of)
-> decltype(source.op(std::forward<OperatorFactory>(of))) {
	return	source.op(std::forward<OperatorFactory>(of));
}

#endif


#if !defined(RXCPP_RX_GROUPED_OBSERVABLE_HPP)
#define RXCPP_RX_GROUPED_OBSERVABLE_HPP

//_include "rx-includes.hpp"

namespace rxcpp {

	namespace detail {

		template<class K, class Source>
		struct has_on_get_key_for
		{
			struct not_void {};
			template<class CS>
			static auto check(int) -> decltype((*(CS*)nullptr).on_get_key());
			template<class CS>
			static not_void check(...);

			typedef decltype(check<Source>(0)) detail_result;
			static const bool value = std::is_same<detail_result, rxu::decay_t<K>>::value;
		};

	}

	template<class K, class T>
	class dynamic_grouped_observable
		: public dynamic_observable<T>
	{
	public:
		typedef rxu::decay_t<K> key_type;
		typedef tag_dynamic_grouped_observable dynamic_observable_tag;

	private:
		struct state_type
			: public std::enable_shared_from_this<state_type>
		{
			typedef std::function<key_type()> ongetkey_type;

			ongetkey_type on_get_key;
		};
		std::shared_ptr<state_type> state;

		template<class U, class V>
		friend bool operator==(const dynamic_grouped_observable<U, V>&, const dynamic_grouped_observable<U, V>&);

		template<class U, class V>
		void construct(const dynamic_grouped_observable<U, V>& o, const tag_dynamic_grouped_observable&) {
			state = o.state;
		}

		template<class U, class V>
		void construct(dynamic_grouped_observable<U, V>&& o, const tag_dynamic_grouped_observable&) {
			state = std::move(o.state);
		}

		template<class SO>
		void construct(SO&& source, const rxs::tag_source&) {
			auto so = std::make_shared<rxu::decay_t<SO>>(std::forward<SO>(source));
			state->on_get_key = [so]() mutable {
				return so->on_get_key();
			};
		}

	public:

		dynamic_grouped_observable()
		{
		}
```

```
                template<class SOF>
                explicit dynamic_grouped_observable(SOF sof)
                        : dynamic_observable<T>(sof)
                        , state(std::make_shared<state_type>())
                {
                                construct(std::move(sof),
                                        typename std::conditional<is_dynamic_grouped_observable<SOF>::value,
tag_dynamic_grouped_observable, rxs::tag_source>::type());
                        }

                template<class SF, class CF>
                dynamic_grouped_observable(SF&& sf, CF&& cf)
                        : dynamic_observable<T>(std::forward<SF>(sf))
                        , state(std::make_shared<state_type>())
                {
                                state->on_connect = std::forward<CF>(cf);
                        }

                using dynamic_observable<T>::on_subscribe;

                key_type on_get_key() const {
                        return state->on_get_key();
                }
        };

        template<class K, class T>
        inline bool operator==(const dynamic_grouped_observable<K, T>& lhs, const dynamic_grouped_observable<K, T>& rhs) {
                return lhs.state == rhs.state;
        }
        template<class K, class T>
        inline bool operator!=(const dynamic_grouped_observable<K, T>& lhs, const dynamic_grouped_observable<K, T>& rhs) {
                return !(lhs == rhs);
        }

        template<class K, class T, class Source>
        grouped_observable<K, T> make_dynamic_grouped_observable(Source&& s) {
                return grouped_observable<K, T>(dynamic_grouped_observable<K, T>(std::forward<Source>(s)));
        }



        /*!
        \brief a source of observables which each emit values from one category specified by the key selector.

        \ingroup group-observable

        */
        template<class K, class T, class SourceOperator>
        class grouped_observable
                : public observable<T, SourceOperator>
        {
                typedef grouped_observable<K, T, SourceOperator> this_type;
                typedef observable<T, SourceOperator> base_type;
                typedef rxu::decay_t<SourceOperator> source_operator_type;

                static_assert(detail::has_on_get_key_for<K, source_operator_type>::value, "inner must have on_get_key method
key_type()");

        public:
                typedef rxu::decay_t<K> key_type;
                typedef tag_grouped_observable observable_tag;

                grouped_observable()
                {
                }

                explicit grouped_observable(const SourceOperator& o)
                        : base_type(o)
                {
                }
                explicit grouped_observable(SourceOperator&& o)
                        : base_type(std::move(o))
                {
                }

                // implicit conversion between observables of the same value_type
                template<class SO>
                grouped_observable(const grouped_observable<K, T, SO>& o)
                        : base_type(o)
                {}
                // implicit conversion between observables of the same value_type
                template<class SO>
```

```
                grouped_observable(grouped_observable<K, T, SO>&& o)
                        : base_type(std::move(o))
                {}

                ///
                /// performs type-forgetting conversion to a new grouped_observable
                ///
                grouped_observable<K, T> as_dynamic() const {
                        return *this;
                }

                key_type get_key() const {
                        return base_type::source_operator.on_get_key();
                }
        };


}

//
// support range() >> filter() >> subscribe() syntax
// '>>' is spelled 'stream'
//
template<class K, class T, class SourceOperator, class OperatorFactory>
auto operator >> (const rxcpp::grouped_observable<K, T, SourceOperator>& source, OperatorFactory&& of)
-> decltype(source.op(std::forward<OperatorFactory>(of))) {
        return  source.op(std::forward<OperatorFactory>(of));
}

//
// support range() | filter() | subscribe() syntax
// '|' is spelled 'pipe'
//
template<class K, class T, class SourceOperator, class OperatorFactory>
auto operator | (const rxcpp::grouped_observable<K, T, SourceOperator>& source, OperatorFactory&& of)
-> decltype(source.op(std::forward<OperatorFactory>(of))) {
        return  source.op(std::forward<OperatorFactory>(of));
}

#endif


#if !defined(RXCPP_LITE)

//#pragma once

/*! \file rx-all.hpp

\brief Returns an Observable that emits true if every item emitted by the source Observable satisfies a specified condition, otherwise false.
Emits true if the source Observable terminates without emitting any item.

\tparam Predicate  the type of the test function.

\param p  the test function to test items emitted by the source Observable.

\return  Observable that emits true if every item emitted by the source observable satisfies a specified condition, otherwise false.

\sample
\snippet all.cpp all sample
\snippet output.txt all sample
*/

#if !defined(RXCPP_OPERATORS_RX_ALL_HPP)
#define RXCPP_OPERATORS_RX_ALL_HPP

// include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class... AN>
                        struct all_invalid_arguments {};

                        template<class... AN>
                        struct all_invalid : public rxo::operator_base<all_invalid_arguments<AN...>> {
                                using type = observable<all_invalid_arguments<AN...>, all_invalid<AN...>>;
                        };
                        template<class... AN>
                        using all_invalid_t = typename all_invalid<AN...>::type;
```

```
                template<class T, class Predicate>
                struct all
                {
                        typedef rxu::decay_t<T> source_value_type;
                        typedef rxu::decay_t<Predicate> test_type;
                        test_type test;

                        typedef bool value_type;

                        all(test_type t)
                                : test(std::move(t))
                        {
                        }

                        template<class Subscriber>
                        struct all_observer
                        {
                                typedef all_observer<Subscriber> this_type;
                                typedef source_value_type value_type;
                                typedef rxu::decay_t<Subscriber> dest_type;
                                typedef observer<value_type, this_type> observer_type;
                                dest_type dest;
                                test_type test;
                                mutable bool done;

                                all_observer(dest_type d, test_type t)
                                        : dest(std::move(d))
                                        , test(std::move(t)),
                                        done(false)
                                {
                                }
                                void on_next(source_value_type v) const {
                                        auto filtered = on_exception([&]() {
                                                return !this->test(v); },
                                                        dest);
                                                if (filtered.empty()) {
                                                        return;
                                                }
                                                if (filtered.get() && !done) {
                                                        done = true;
                                                        dest.on_next(false);
                                                        dest.on_completed();
                                                }
                                }
                                void on_error(std::exception_ptr e) const {
                                        dest.on_error(e);
                                }
                                void on_completed() const {
                                        if (!done) {
                                                done = true;
                                                dest.on_next(true);
                                                dest.on_completed();
                                        }
                                }

                                static subscriber<value_type, observer_type> make(dest_type d, test_type t) {
                                        return make_subscriber<value_type>(d, this_type(d, std::move(t)));
                                }
                        };

                        template<class Subscriber>
                        auto operator()(Subscriber dest) const
                                -> decltype(all_observer<Subscriber>::make(std::move(dest), test)) {
                                return      all_observer<Subscriber>::make(std::move(dest), test);
                        }
                };

        }

        /*! @copydoc rx-all.hpp
        */
        template<class... AN>
        auto all(AN&&... an)
                ->     operator_factory<all_tag, AN...> {
                return operator_factory<all_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
        }

}

template<>
struct member_overload<all_tag>
```

```
	{
		template<class Observable, class Predicate,
		class SourceValue = rxu::value_type_t<Observable>,
		class Enabled = rxu::enable_if_all_true_type_t<
			is_observable<Observable >> ,
		class All = rxo::detail::all<SourceValue, rxu::decay_t<Predicate>>,
		class Value = rxu::value_type_t<All >>
			static auto member(Observable&& o, Predicate&& p)
			-> decltype(o.template lift<Value>(All(std::forward<Predicate>(p)))) {
				return	o.template lift<Value>(All(std::forward<Predicate>(p)));
			}

		template<class... AN>
		static operators::detail::all_invalid_t<AN...> member(const AN&...) {
			std::terminate();
			return{};
			static_assert(sizeof...(AN) == 10000, "all takes (Predicate)");
		}
	};

}

#endif


/*! \file rx-any.hpp

\brief Returns an Observable that emits true if any item emitted by the source Observable satisfies a specified condition, otherwise false. Emits false if the source Observable terminates without emitting any item.

\tparam Predicate the type of the test function.

\param p the test function to test items emitted by the source Observable.

\return  An observable that emits true if any item emitted by the source observable satisfies a specified condition, otherwise false.

Some basic any- operators have already been implemented:
- rxcpp::operators::exists
- rxcpp::operators::contains

\sample
\snippet exists.cpp exists sample
\snippet output.txt exists sample

\sample
\snippet contains.cpp contains sample
\snippet output.txt contains sample
*/


#if !defined(RXCPP_OPERATORS_RX_ANY_HPP)
#define RXCPP_OPERATORS_RX_ANY_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace operators {

		namespace detail {

			template<class... AN>
			struct any_invalid_arguments {};

			template<class... AN>
			struct any_invalid : public rxo::operator_base<any_invalid_arguments<AN...>> {
				using type = observable<any_invalid_arguments<AN...>, any_invalid<AN...>>;
			};
			template<class... AN>
			using any_invalid_t = typename any_invalid<AN...>::type;

			template<class T, class Predicate>
			struct any
			{
				typedef rxu::decay_t<T> source_value_type;
				typedef bool value_type;
				typedef rxu::decay_t<Predicate> test_type;
				test_type test;

				any(test_type t)
					: test(std::move(t))
				{
```

```
                                        }

                                        template<class Subscriber>
                                        struct any_observer
                                        {
                                                typedef any_observer<Subscriber> this_type;
                                                typedef source_value_type value_type;
                                                typedef rxu::decay_t<Subscriber> dest_type;
                                                typedef observer<value_type, this_type> observer_type;
                                                dest_type dest;
                                                test_type test;
                                                mutable bool done;

                                                any_observer(dest_type d, test_type t)
                                                        : dest(std::move(d))
                                                        , test(std::move(t)),
                                                        done(false)
                                                {
                                                }
                                                void on_next(source_value_type v) const {
                                                        auto filtered = on_exception([&]() {
                                                                return !this->test(v); },
                                                                        dest);
                                                                if (filtered.empty()) {
                                                                        return;
                                                                }
                                                                if (!filtered.get() && !done) {
                                                                        done = true;
                                                                        dest.on_next(true);
                                                                        dest.on_completed();
                                                                }
                                                }
                                                void on_error(std::exception_ptr e) const {
                                                        dest.on_error(e);
                                                }
                                                void on_completed() const {
                                                        if (!done) {
                                                                done = true;
                                                                dest.on_next(false);
                                                                dest.on_completed();
                                                        }
                                                }

                                                static subscriber<value_type, observer_type> make(dest_type d, test_type t) {
                                                        return make_subscriber<value_type>(d, this_type(d, std::move(t)));
                                                }
                                        };

                                        template<class Subscriber>
                                        auto operator()(Subscriber dest) const
                                                -> decltype(any_observer<Subscriber>::make(std::move(dest), test)) {
                                                return     any_observer<Subscriber>::make(std::move(dest), test);
                                        }
                                };

                        }

                        /*! @copydoc rx-any.hpp
                        */
                        template<class... AN>
                        auto any(AN&&... an)
                                ->    operator_factory<any_tag, AN...> {
                                return operator_factory<any_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
                        }

                        /*! \brief Returns an Observable that emits true if any item emitted by the source Observable satisfies a specified condition, otherwise false. Emits false if the source Observable terminates without emitting any item.

                        \tparam Predicate the type of the test function.

                        \param p the test function to test items emitted by the source Observable.

                        \return  An observable that emits true if any item emitted by the source observable satisfies a specified condition, otherwise false.

                        \sample
                        \snippet exists.cpp exists sample
                        \snippet output.txt exists sample
                        */
                        template<class... AN>
                        auto exists(AN&&... an)
                                ->    operator_factory<exists_tag, AN...> {
```

```
                return operator_factory<exists_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
            }

            /*! \brief Returns an Observable that emits true if the source Observable emitted a specified item, otherwise false. Emits false if the source Observable terminates without emitting any item.

            \tparam T the type of the item to search for.

            \param value the item to search for.

            \return An observable that emits true if the source Observable emitted a specified item, otherwise false.

            \sample
            \snippet contains.cpp contains sample
            \snippet output.txt contains sample
            */
            template<class... AN>
            auto contains(AN&&... an)
                ->     operator_factory<contains_tag, AN...> {
                return operator_factory<contains_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
            }

        }

        template<>
        struct member_overload<any_tag>
        {
            template<class Observable, class Predicate,
            class SourceValue = rxu::value_type_t<Observable>,
            class Enabled = rxu::enable_if_all_true_type_t<
                is_observable<Observable >> ,
            class Any = rxo::detail::any<SourceValue, rxu::decay_t<Predicate>>,
            class Value = rxu::value_type_t<Any >>
                static auto member(Observable&& o, Predicate&& p)
                -> decltype(o.template lift<Value>(Any(std::forward<Predicate>(p)))) {
                    return  o.template lift<Value>(Any(std::forward<Predicate>(p)));
                }

            template<class... AN>
            static operators::detail::any_invalid_t<AN...> member(const AN&...) {
                std::terminate();
                return{};
                static_assert(sizeof...(AN) == 10000, "any takes (Predicate)");
            }
        };

        template<>
        struct member_overload<exists_tag>
            : member_overload<any_tag>
        {
            using member_overload<any_tag>::member;

            template<class... AN>
            static operators::detail::any_invalid_t<AN...> member(const AN&...) {
                std::terminate();
                return{};
                static_assert(sizeof...(AN) == 10000, "exists takes (Predicate)");
            }
        };

        template<>
        struct member_overload<contains_tag>
        {
            template<class Observable, class T,
            class SourceValue = rxu::value_type_t<Observable>,
            class Enabled = rxu::enable_if_all_true_type_t<
                is_observable<Observable >> ,
            class Predicate = std::function<bool(T)>,
            class Any = rxo::detail::any<SourceValue, rxu::decay_t<Predicate>>,
            class Value = rxu::value_type_t<Any >>
                static auto member(Observable&& o, T&& value)
                -> decltype(o.template lift<Value>(Any(nullptr))) {
                    return  o.template lift<Value>(Any([value](T n) { return n == value; }));
                }

            template<class... AN>
            static operators::detail::any_invalid_t<AN...> member(const AN&...) {
                std::terminate();
                return{};
                static_assert(sizeof...(AN) == 10000, "contains takes (T)");
            }
        };
```

```
}

#endif


/*! \file rx-combine_latest.hpp

\brief For each item from all of the observables select a value to emit from the new observable that is returned.

\tparam AN  types of scheduler (optional), aggregate function (optional), and source observables

\param  an  scheduler (optional), aggregation function (optional), and source observables

\return  Observable that emits items that are the result of combining the items emitted by the source observables.

If scheduler is omitted, identity_current_thread is used.

If aggregation function is omitted, the resulting observable returns tuples of emitted items.

\sample

Neither scheduler nor aggregation function are present:
\snippet combine_latest.cpp combine_latest sample
\snippet output.txt combine_latest sample

Only scheduler is present:
\snippet combine_latest.cpp Coordination combine_latest sample
\snippet output.txt Coordination combine_latest sample

Only aggregation function is present:
\snippet combine_latest.cpp Selector combine_latest sample
\snippet output.txt Selector combine_latest sample

Both scheduler and aggregation function are present:
\snippet combine_latest.cpp Coordination+Selector combine_latest sample
\snippet output.txt Coordination+Selector combine_latest sample
*/

#if !defined(RXCPP_OPERATORS_RX_COMBINE_LATEST_HPP)
#define RXCPP_OPERATORS_RX_COMBINE_LATEST_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace operators {

		namespace detail {

			template<class... AN>
			struct combine_latest_invalid_arguments {};

			template<class... AN>
			struct combine_latest_invalid : public rxo::operator_base<combine_latest_invalid_arguments<AN...>> {
				using type = observable<combine_latest_invalid_arguments<AN...>,
combine_latest_invalid<AN...>>;
			};
			template<class... AN>
			using combine_latest_invalid_t = typename combine_latest_invalid<AN...>::type;

			template<class Selector, class... ObservableN>
			struct is_combine_latest_selector_check {
				typedef rxu::decay_t<Selector> selector_type;

				struct tag_not_valid;
				template<class CS, class... CON>
				static auto check(int) -> decltype((*(CS*)nullptr)((*(typename CON::value_type*)nullptr)...));
				template<class CS, class... CON>
				static tag_not_valid check(...);

				using type = decltype(check<selector_type, rxu::decay_t<ObservableN>...>(0));

				static const bool value = !std::is_same<type, tag_not_valid>::value;
			};

			template<class Selector, class... ObservableN>
			struct invalid_combine_latest_selector {
				static const bool value = false;
			};

			template<class Selector, class... ObservableN>
```

```
struct is_combine_latest_selector : public std::conditional<
	is_combine_latest_selector_check<Selector, ObservableN...>::value,
	is_combine_latest_selector_check<Selector, ObservableN...>,
	invalid_combine_latest_selector<Selector, ObservableN... >> ::type {
};

template<class Selector, class... ON>
using result_combine_latest_selector_t = typename is_combine_latest_selector<Selector, ON...>::type;

template<class Coordination, class Selector, class... ObservableN>
struct combine_latest_traits {

	typedef std::tuple<ObservableN...> tuple_source_type;
	typedef std::tuple<rxu::detail::maybe<typename ObservableN::value_type>...>
tuple_source_value_type;


	typedef rxu::decay_t<Selector> selector_type;
	typedef rxu::decay_t<Coordination> coordination_type;

	typedef typename is_combine_latest_selector<selector_type, ObservableN...>::type value_type;
};

template<class Coordination, class Selector, class... ObservableN>
struct combine_latest : public operator_base<rxu::value_type_t<combine_latest_traits<Coordination, Selector,
ObservableN...>>>
{
	typedef combine_latest<Coordination, Selector, ObservableN...> this_type;

	typedef combine_latest_traits<Coordination, Selector, ObservableN...> traits;

	typedef typename traits::tuple_source_type tuple_source_type;
	typedef typename traits::tuple_source_value_type tuple_source_value_type;

	typedef typename traits::selector_type selector_type;

	typedef typename traits::coordination_type coordination_type;
	typedef typename coordination_type::coordinator_type coordinator_type;

	struct values
	{
		values(tuple_source_type o, selector_type s, coordination_type sf)
		: source(std::move(o))
		, selector(std::move(s))
		, coordination(std::move(sf))
		{
		}
		tuple_source_type source;
		selector_type selector;
		coordination_type coordination;
	};
	values initial;

	combine_latest(coordination_type sf, selector_type s, tuple_source_type ts)
		: initial(std::move(ts), std::move(s), std::move(sf))
	{
	}

	template<int Index, class State>
	void subscribe_one(std::shared_ptr<State> state) const {

		typedef typename std::tuple_element<Index, tuple_source_type>::type::value_type
source_value_type;


		composite_subscription innercs;

		// when the out observer is unsubscribed all the
		// inner subscriptions are unsubscribed as well
		state->out.add(innercs);

		auto source = on_exception(
			[&](){return state->coordinator.in(std::get<Index>(state->source)); },
			state->out);
		if (source.empty()) {
			return;
		}

		// this subscribe does not share the observer subscription
		// so that when it is unsubscribed the observer can be called
		// until the inner subscriptions have finished
		auto sink = make_subscriber<source_value_type>(
			state->out,
			innercs,
```

```
                                                // on_next
                                                [state](source_value_type st) {
                                                auto& value = std::get<Index>(state->latest);

                                                if (value.empty()) {
                                                        ++state->valuesSet;
                                                }

                                                value.reset(st);

                                                if (state->valuesSet == sizeof... (ObservableN)) {
                                                        auto values = rxu::surely(state->latest);
                                                        auto selectedResult = rxu::apply(values, state->selector);
                                                        state->out.on_next(selectedResult);
                                                }
                                        },
                                                // on_error
                                                [state](std::exception_ptr e) {
                                                state->out.on_error(e);
                                        },
                                                // on_completed
                                                [state]() {
                                                if (--state->pendingCompletions == 0) {
                                                        state->out.on_completed();
                                                }
                                        }
                                        );
                                        auto selectedSink = on_exception(
                                                [&](){return state->coordinator.out(sink); },
                                                state->out);
                                        if (selectedSink.empty()) {
                                                return;
                                        }
                                        source->subscribe(std::move(selectedSink.get()));
                                }
                                template<class State, int... IndexN>
                                void subscribe_all(std::shared_ptr<State> state, rxu::values<int, IndexN...>) const {
                                        bool subscribed[] = { (subscribe_one<IndexN>(state), true)... };
                                        subscribed[0] = (*subscribed); // silence warning
                                }

                                template<class Subscriber>
                                void on_subscribe(Subscriber scbr) const {
                                        static_assert(is_subscriber<Subscriber>::value, "subscribe must be passed a subscriber");

                                        typedef Subscriber output_type;

                                        struct combine_latest_state_type
                                                : public std::enable_shared_from_this<combine_latest_state_type>
                                                , public values
                                        {
                                                combine_latest_state_type(values i, coordinator_type coor, output_type oarg)
                                                : values(std::move(i))
                                                , pendingCompletions(sizeof... (ObservableN))
                                                , valuesSet(0)
                                                , coordinator(std::move(coor))
                                                , out(std::move(oarg))
                                                {
                                                }

                                                // on_completed on the output must wait until all the
                                                // subscriptions have received on_completed
                                                mutable int pendingCompletions;
                                                mutable int valuesSet;
                                                mutable tuple_source_value_type latest;
                                                coordinator_type coordinator;
                                                output_type out;
                                        };

                                        auto coordinator = initial.coordination.create_coordinator(scbr.get_subscription());

                                        // take a copy of the values for each subscription
                                        auto state = std::make_shared<combine_latest_state_type>(initial, std::move(coordinator),
std::move(scbr));

                                        subscribe_all(state, typename rxu::values_from<int, sizeof...(ObservableN)>::type());
                                }
                        };

                }

                /*! @copydoc rx-combine_latest.hpp
```

```
                        */
                        template<class... AN>
                        auto combine_latest(AN&&... an)
                                ->      operator_factory<combine_latest_tag, AN...> {
                                return operator_factory<combine_latest_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
                        }

        }

        template<>
        struct member_overload<combine_latest_tag>
        {
                template<class Observable, class... ObservableN,
                class Enabled = rxu::enable_if_all_true_type_t<
                        all_observables<Observable, ObservableN... >> ,
                class combine_latest = rxo::detail::combine_latest<identity_one_worker, rxu::detail::pack, rxu::decay_t<Observable>,
rxu::decay_t<ObservableN>...>,
                class Value = rxu::value_type_t<combine_latest>,
                class Result = observable<Value, combine_latest >>
                        static Result member(Observable&& o, ObservableN&&... on)
                {
                                        return Result(combine_latest(identity_current_thread(), rxu::pack(),
std::make_tuple(std::forward<Observable>(o), std::forward<ObservableN>(on)...)));
                        }

                template<class Observable, class Selector, class... ObservableN,
                class Enabled = rxu::enable_if_all_true_type_t<
                        operators::detail::is_combine_latest_selector<Selector, Observable, ObservableN...>,
                        all_observables<Observable, ObservableN... >> ,
                class ResolvedSelector = rxu::decay_t<Selector>,
                class combine_latest = rxo::detail::combine_latest<identity_one_worker, ResolvedSelector, rxu::decay_t<Observable>,
rxu::decay_t<ObservableN>...>,
                class Value = rxu::value_type_t<combine_latest>,
                class Result = observable<Value, combine_latest >>
                        static Result member(Observable&& o, Selector&& s, ObservableN&&... on)
                {
                                        return Result(combine_latest(identity_current_thread(), std::forward<Selector>(s),
std::make_tuple(std::forward<Observable>(o), std::forward<ObservableN>(on)...)));
                        }

                template<class Coordination, class Observable, class... ObservableN,
                class Enabled = rxu::enable_if_all_true_type_t<
                        is_coordination<Coordination>,
                        all_observables<Observable, ObservableN... >> ,
                class combine_latest = rxo::detail::combine_latest<Coordination, rxu::detail::pack, rxu::decay_t<Observable>,
rxu::decay_t<ObservableN>...>,
                class Value = rxu::value_type_t<combine_latest>,
                class Result = observable<Value, combine_latest >>
                        static Result member(Observable&& o, Coordination&& cn, ObservableN&&... on)
                {
                                        return Result(combine_latest(std::forward<Coordination>(cn), rxu::pack(),
std::make_tuple(std::forward<Observable>(o), std::forward<ObservableN>(on)...)));
                        }

                template<class Coordination, class Selector, class Observable, class... ObservableN,
                class Enabled = rxu::enable_if_all_true_type_t<
                        is_coordination<Coordination>,
                        operators::detail::is_combine_latest_selector<Selector, Observable, ObservableN...>,
                        all_observables<Observable, ObservableN... >> ,
                class ResolvedSelector = rxu::decay_t<Selector>,
                class combine_latest = rxo::detail::combine_latest<Coordination, ResolvedSelector, rxu::decay_t<Observable>,
rxu::decay_t<ObservableN>...>,
                class Value = rxu::value_type_t<combine_latest>,
                class Result = observable<Value, combine_latest >>
                        static Result member(Observable&& o, Coordination&& cn, Selector&& s, ObservableN&&... on)
                {
                                        return Result(combine_latest(std::forward<Coordination>(cn), std::forward<Selector>(s),
std::make_tuple(std::forward<Observable>(o), std::forward<ObservableN>(on)...)));
                        }

                template<class... AN>
                static operators::detail::combine_latest_invalid_t<AN...> member(const AN&...) {
                        std::terminate();
                        return{};
                        static_assert(sizeof...(AN) == 10000, "combine_latest takes (optional Coordination, optional Selector, required
Observable, optional Observable...), Selector takes (Observable::value_type...)");
                }
        };

}

#endif
```

```
/*! \file rx-debounce.hpp

\brief  Return an observable that emits an item if a particular timespan has passed without emitting another item from the source observable.

\tparam Duration      the type of the time interval
\tparam Coordination  the type of the scheduler

\param period        the period of time to suppress any emitted items
\param coordination  the scheduler to manage timeout for each event

\return  Observable that emits an item if a particular timespan has passed without emitting another item from the source observable.

\sample
\snippet debounce.cpp debounce sample
\snippet output.txt debounce sample
*/

#if !defined(RXCPP_OPERATORS_RX_DEBOUNCE_HPP)
#define RXCPP_OPERATORS_RX_DEBOUNCE_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class... AN>
                        struct debounce_invalid_arguments {};

                        template<class... AN>
                        struct debounce_invalid : public rxo::operator_base<debounce_invalid_arguments<AN...>> {
                                using type = observable<debounce_invalid_arguments<AN...>, debounce_invalid<AN...>>;
                        };
                        template<class... AN>
                        using debounce_invalid_t = typename debounce_invalid<AN...>::type;

                        template<class T, class Duration, class Coordination>
                        struct debounce
                        {
                                typedef rxu::decay_t<T> source_value_type;
                                typedef rxu::decay_t<Coordination> coordination_type;
                                typedef typename coordination_type::coordinator_type coordinator_type;
                                typedef rxu::decay_t<Duration> duration_type;

                                struct debounce_values
                                {
                                        debounce_values(duration_type p, coordination_type c)
                                        : period(p)
                                        , coordination(c)
                                        {
                                        }

                                        duration_type period;
                                        coordination_type coordination;
                                };
                                debounce_values initial;

                                debounce(duration_type period, coordination_type coordination)
                                        : initial(period, coordination)
                                {
                                }

                                template<class Subscriber>
                                struct debounce_observer
                                {
                                        typedef debounce_observer<Subscriber> this_type;
                                        typedef rxu::decay_t<T> value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<T, this_type> observer_type;

                                        struct debounce_subscriber_values : public debounce_values
                                        {
                                                debounce_subscriber_values(composite_subscription cs, dest_type d,
debounce_values v, coordinator_type c)
                                                : debounce_values(v)
                                                , cs(std::move(cs))
                                                , dest(std::move(d))
                                                , coordinator(std::move(c))
```

```
                                                , worker(coordinator.get_worker())
                                                , index(0)
                                                {
                                                }

                                                composite_subscription cs;
                                                dest_type dest;
                                                coordinator_type coordinator;
                                                rxsc::worker worker;
                                                mutable std::size_t index;
                                                mutable rxu::maybe<value_type> value;
                                        };
                                        typedef std::shared_ptr<debounce_subscriber_values> state_type;
                                        state_type state;

                                        debounce_observer(composite_subscription cs, dest_type d, debounce_values v,
coordinator_type c)
                                                :
state(std::make_shared<debounce_subscriber_values>(debounce_subscriber_values(std::move(cs), std::move(d), v, std::move(c))))
                                        {
                                                auto localState = state;

                                                auto disposer = [=](const rxsc::schedulable&){
                                                        localState->cs.unsubscribe();
                                                        localState->dest.unsubscribe();
                                                        localState->worker.unsubscribe();
                                                };
                                                auto selectedDisposer = on_exception(
                                                        [&](){ return localState->coordinator.act(disposer); },
                                                        localState->dest);
                                                if (selectedDisposer.empty()) {
                                                        return;
                                                }

                                                localState->dest.add([=](){
                                                        localState->worker.schedule(selectedDisposer.get());
                                                });
                                                localState->cs.add([=](){
                                                        localState->worker.schedule(selectedDisposer.get());
                                                });
                                        }

                                        static std::function<void(const rxsc::schedulable&)> produce_item(std::size_t id,
state_type state) {
                                                auto produce = [id, state](const rxsc::schedulable&) {
                                                        if (id != state->index)
                                                                return;

                                                        state->dest.on_next(*state->value);
                                                        state->value.reset();
                                                };

                                                auto selectedProduce = on_exception(
                                                        [&](){ return state->coordinator.act(produce); },
                                                        state->dest);
                                                if (selectedProduce.empty()) {
                                                        return std::function<void(const rxsc::schedulable&)>();
                                                }

                                                return std::function<void(const rxsc::schedulable&)>(selectedProduce.get());
                                        }

                                        void on_next(T v) const {
                                                auto localState = state;
                                                auto work = [v, localState](const rxsc::schedulable&) {
                                                        auto new_id = ++localState->index;
                                                        auto produce_time = localState->worker.now() + localState->period;

                                                        localState->value.reset(v);
                                                        localState->worker.schedule(produce_time, produce_item(new_id,
localState));
                                                };
                                                auto selectedWork = on_exception(
                                                        [&](){return localState->coordinator.act(work); },
                                                        localState->dest);
                                                if (selectedWork.empty()) {
                                                        return;
                                                }
                                                localState->worker.schedule(selectedWork.get());
                                        }

                                        void on_error(std::exception_ptr e) const {
```

```
                                                auto localState = state;
                                                auto work = [e, localState](const rxsc::schedulable&) {
                                                        localState->dest.on_error(e);
                                                        localState->value.reset();
                                                };
                                                auto selectedWork = on_exception(
                                                        [&](){ return localState->coordinator.act(work); },
                                                        localState->dest);
                                                if (selectedWork.empty()) {
                                                        return;
                                                }
                                                localState->worker.schedule(selectedWork.get());
                                        }

                                        void on_completed() const {
                                                auto localState = state;
                                                auto work = [localState](const rxsc::schedulable&) {
                                                        if (!localState->value.empty()) {
                                                                localState->dest.on_next(*localState->value);
                                                        }
                                                        localState->dest.on_completed();
                                                };
                                                auto selectedWork = on_exception(
                                                        [&](){ return localState->coordinator.act(work); },
                                                        localState->dest);
                                                if (selectedWork.empty()) {
                                                        return;
                                                }
                                                localState->worker.schedule(selectedWork.get());
                                        }

                                        static subscriber<T, observer_type> make(dest_type d, debounce_values v) {
                                                auto cs = composite_subscription();
                                                auto coordinator = v.coordination.create_coordinator();

                                                return make_subscriber<T>(cs, observer_type(this_type(cs, std::move(d),
std::move(v), std::move(coordinator))));
                                        }
                                };

                                template<class Subscriber>
                                auto operator()(Subscriber dest) const
                                        -> decltype(debounce_observer<Subscriber>::make(std::move(dest), initial)) {
                                        return      debounce_observer<Subscriber>::make(std::move(dest), initial);
                                }
                        };

                }

                /*! @copydoc rx-debounce.hpp
                */
                template<class... AN>
                auto debounce(AN&&... an)
                        ->      operator_factory<debounce_tag, AN...> {
                        return operator_factory<debounce_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
                }

        }

        template<>
        struct member_overload<debounce_tag>
        {
                template<class Observable, class Duration,
                class Enabled = rxu::enable_if_all_true_type_t<
                        is_observable<Observable>,
                        rxu::is_duration<Duration >> ,
                class SourceValue = rxu::value_type_t<Observable>,
                class Debounce = rxo::detail::debounce<SourceValue, rxu::decay_t<Duration>, identity_one_worker >>
                        static auto member(Observable&& o, Duration&& d)
                        -> decltype(o.template lift<SourceValue>(Debounce(std::forward<Duration>(d), identity_current_thread()))) {
                                return      o.template lift<SourceValue>(Debounce(std::forward<Duration>(d),
identity_current_thread()));
                        }

                template<class Observable, class Coordination, class Duration,
                class Enabled = rxu::enable_if_all_true_type_t<
                        is_observable<Observable>,
                        is_coordination<Coordination>,
                        rxu::is_duration<Duration >> ,
                class SourceValue = rxu::value_type_t<Observable>,
                class Debounce = rxo::detail::debounce<SourceValue, rxu::decay_t<Duration>, rxu::decay_t<Coordination>>>
                        static auto member(Observable&& o, Coordination&& cn, Duration&& d)
```

```
                -> decltype(o.template lift<SourceValue>(Debounce(std::forward<Duration>(d),
std::forward<Coordination>(cn)))) {
                        return      o.template lift<SourceValue>(Debounce(std::forward<Duration>(d),
std::forward<Coordination>(cn)));
                }

        template<class Observable, class Coordination, class Duration,
        class Enabled = rxu::enable_if_all_true_type_t<
                is_observable<Observable>,
                is_coordination<Coordination>,
                rxu::is_duration<Duration >> ,
        class SourceValue = rxu::value_type_t<Observable>,
        class Debounce = rxo::detail::debounce<SourceValue, rxu::decay_t<Duration>, rxu::decay_t<Coordination>>>
                static auto member(Observable&& o, Duration&& d, Coordination&& cn)
                -> decltype(o.template lift<SourceValue>(Debounce(std::forward<Duration>(d),
std::forward<Coordination>(cn)))) {
                        return      o.template lift<SourceValue>(Debounce(std::forward<Duration>(d),
std::forward<Coordination>(cn)));
                }

        template<class... AN>
        static operators::detail::debounce_invalid_t<AN...> member(const AN&...) {
                std::terminate();
                return{};
                static_assert(sizeof...(AN) == 10000, "debounce takes (optional Coordination, required Duration) or (required
Duration, optional Coordination)");
        }
    };

}

#endif


/*! \file rx-delay.hpp

\brief Return an observable that emits each item emitted by the source observable after the specified delay.

\tparam Duration      the type of time interval
\tparam Coordination  the type of the scheduler

\param period       the period of time each item is delayed
\param coordination  the scheduler for the delays

\return  Observable that emits each item emitted by the source observable after the specified delay.

\sample
\snippet delay.cpp delay period+coordination sample
\snippet output.txt delay period+coordination sample
*/

#if !defined(RXCPP_OPERATORS_RX_DELAY_HPP)
#define RXCPP_OPERATORS_RX_DELAY_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

    namespace operators {

        namespace detail {

            template<class... AN>
            struct delay_invalid_arguments {};

            template<class... AN>
            struct delay_invalid : public rxo::operator_base<delay_invalid_arguments<AN...>> {
                using type = observable<delay_invalid_arguments<AN...>, delay_invalid<AN...>>;
            };
            template<class... AN>
            using delay_invalid_t = typename delay_invalid<AN...>::type;

            template<class T, class Duration, class Coordination>
            struct delay
            {
                typedef rxu::decay_t<T> source_value_type;
                typedef rxu::decay_t<Coordination> coordination_type;
                typedef typename coordination_type::coordinator_type coordinator_type;
                typedef rxu::decay_t<Duration> duration_type;

                struct delay_values
                {
```

```
                                        delay_values(duration_type p, coordination_type c)
                                        : period(p)
                                        , coordination(c)
                                        {
                                        }
                                        duration_type period;
                                        coordination_type coordination;
                                };
                                delay_values initial;

                                delay(duration_type period, coordination_type coordination)
                                        : initial(period, coordination)
                                {
                                }

                                template<class Subscriber>
                                struct delay_observer
                                {
                                        typedef delay_observer<Subscriber> this_type;
                                        typedef rxu::decay_t<T> value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<T, this_type> observer_type;

                                        struct delay_subscriber_values : public delay_values
                                        {
                                                delay_subscriber_values(composite_subscription cs, dest_type d, delay_values
v, coordinator_type c)
                                                : delay_values(v)
                                                , cs(std::move(cs))
                                                , dest(std::move(d))
                                                , coordinator(std::move(c))
                                                , worker(coordinator.get_worker())
                                                , expected(worker.now())
                                                {
                                                }
                                                composite_subscription cs;
                                                dest_type dest;
                                                coordinator_type coordinator;
                                                rxsc::worker worker;
                                                rxsc::scheduler::clock_type::time_point expected;
                                        };
                                        std::shared_ptr<delay_subscriber_values> state;

                                        delay_observer(composite_subscription cs, dest_type d, delay_values v, coordinator_type
c)
                                                :
state(std::make_shared<delay_subscriber_values>(delay_subscriber_values(std::move(cs), std::move(d), v, std::move(c))))
                                        {
                                                auto localState = state;

                                                auto disposer = [=](const rxsc::schedulable&){
                                                        localState->cs.unsubscribe();
                                                        localState->dest.unsubscribe();
                                                        localState->worker.unsubscribe();
                                                };
                                                auto selectedDisposer = on_exception(
                                                        [&](){return localState->coordinator.act(disposer); },
                                                        localState->dest);
                                                if (selectedDisposer.empty()) {
                                                        return;
                                                }

                                                localState->dest.add([=](){
                                                        localState->worker.schedule(selectedDisposer.get());
                                                });
                                                localState->cs.add([=](){
                                                        localState->worker.schedule(localState->worker.now() + localState-
>period, selectedDisposer.get());
                                                });
                                        }

                                        void on_next(T v) const {
                                                auto localState = state;
                                                auto work = [v, localState](const rxsc::schedulable&){
                                                        localState->dest.on_next(v);
                                                };
                                                auto selectedWork = on_exception(
                                                        [&](){return localState->coordinator.act(work); },
                                                        localState->dest);
                                                if (selectedWork.empty()) {
                                                        return;
                                                }
```

```
                                                localState->worker.schedule(localState->worker.now() + localState->period,
selectedWork.get());
                                        }

                                        void on_error(std::exception_ptr e) const {
                                                auto localState = state;
                                                auto work = [e, localState](const rxsc::schedulable&){
                                                        localState->dest.on_error(e);
                                                };
                                                auto selectedWork = on_exception(
                                                        [&](){return localState->coordinator.act(work); },
                                                        localState->dest);
                                                if (selectedWork.empty()) {
                                                        return;
                                                }
                                                localState->worker.schedule(selectedWork.get());
                                        }

                                        void on_completed() const {
                                                auto localState = state;
                                                auto work = [localState](const rxsc::schedulable&){
                                                        localState->dest.on_completed();
                                                };
                                                auto selectedWork = on_exception(
                                                        [&](){return localState->coordinator.act(work); },
                                                        localState->dest);
                                                if (selectedWork.empty()) {
                                                        return;
                                                }
                                                localState->worker.schedule(localState->worker.now() + localState->period,
selectedWork.get());
                                        }

                                        static subscriber<T, observer_type> make(dest_type d, delay_values v) {
                                                auto cs = composite_subscription();
                                                auto coordinator = v.coordination.create_coordinator();

                                                return make_subscriber<T>(cs, observer_type(this_type(cs, std::move(d),
std::move(v), std::move(coordinator))));
                                        }
                                };

                                template<class Subscriber>
                                auto operator()(Subscriber dest) const
                                        -> decltype(delay_observer<Subscriber>::make(std::move(dest), initial)) {
                                        return      delay_observer<Subscriber>::make(std::move(dest), initial);
                                }
                        };

                }

                /*! @copydoc rx-delay.hpp
                */
                template<class... AN>
                auto delay(AN&&... an)
                        ->      operator_factory<delay_tag, AN...> {
                        return operator_factory<delay_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
                }

        }

        template<>
        struct member_overload<delay_tag>
        {
                template<class Observable, class Duration,
                class Enabled = rxu::enable_if_all_true_type_t<
                        is_observable<Observable>,
                        rxu::is_duration<Duration >> ,
                class SourceValue = rxu::value_type_t<Observable>,
                class delay = rxo::detail::delay<SourceValue, rxu::decay_t<Duration>, identity_one_worker >>
                        static auto member(Observable&& o, Duration&& d)
                        -> decltype(o.template lift<SourceValue>(delay(std::forward<Duration>(d), identity_current_thread()))) {
                                return      o.template lift<SourceValue>(delay(std::forward<Duration>(d), identity_current_thread()));
                        }

                template<class Observable, class Coordination, class Duration,
                class Enabled = rxu::enable_if_all_true_type_t<
                        is_observable<Observable>,
                        is_coordination<Coordination>,
                        rxu::is_duration<Duration >> ,
                class SourceValue = rxu::value_type_t<Observable>,
                class delay = rxo::detail::delay<SourceValue, rxu::decay_t<Duration>, rxu::decay_t<Coordination>>>
```

```
                                        static auto member(Observable&& o, Coordination&& cn, Duration&& d)
                                        -> decltype(o.template lift<SourceValue>(delay(std::forward<Duration>(d), std::forward<Coordination>(cn)))) {
                                                return      o.template lift<SourceValue>(delay(std::forward<Duration>(d),
std::forward<Coordination>(cn)));
                                        }

                                template<class Observable, class Coordination, class Duration,
                                class Enabled = rxu::enable_if_all_true_type_t<
                                        is_observable<Observable>,
                                        is_coordination<Coordination>,
                                        rxu::is_duration<Duration >> ,
                                class SourceValue = rxu::value_type_t<Observable>,
                                class delay = rxo::detail::delay<SourceValue, rxu::decay_t<Duration>, rxu::decay_t<Coordination>>>
                                        static auto member(Observable&& o, Duration&& d, Coordination&& cn)
                                        -> decltype(o.template lift<SourceValue>(delay(std::forward<Duration>(d), std::forward<Coordination>(cn)))) {
                                                return      o.template lift<SourceValue>(delay(std::forward<Duration>(d),
std::forward<Coordination>(cn)));
                                        }

                                template<class... AN>
                                static operators::detail::delay_invalid_t<AN...> member(const AN&...) {
                                        std::terminate();
                                        return{};
                                        static_assert(sizeof...(AN) == 10000, "delay takes (optional Coordination, required Duration) or (required
Duration, optional Coordination)");
                                }
                };

}

#endif


/*! \file rx-distinct.hpp

\brief For each item from this observable, filter out repeated values and emit only items that have not already been emitted.

\return Observable that emits those items from the source observable that are distinct.

\note istinct keeps an unordered_set<T> of past values. Due to an issue in multiple implementations of std::hash<T>, rxcpp maintains a whitelist
of hashable types. new types can be added by specializing rxcpp::filtered_hash<T>

\sample
\snippet distinct.cpp distinct sample
\snippet output.txt distinct sample
*/

#if !defined(RXCPP_OPERATORS_RX_DISTINCT_HPP)
#define RXCPP_OPERATORS_RX_DISTINCT_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class... AN>
                        struct distinct_invalid_arguments {};

                        template<class... AN>
                        struct distinct_invalid : public rxo::operator_base<distinct_invalid_arguments<AN...>> {
                                using type = observable<distinct_invalid_arguments<AN...>, distinct_invalid<AN...>>;
                        };
                        template<class... AN>
                        using distinct_invalid_t = typename distinct_invalid<AN...>::type;

                        template<class T>
                        struct distinct
                        {
                                typedef rxu::decay_t<T> source_value_type;

                                template<class Subscriber>
                                struct distinct_observer
                                {
                                        typedef distinct_observer<Subscriber> this_type;
                                        typedef source_value_type value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<value_type, this_type> observer_type;
                                        dest_type dest;
```

```
                                                mutable std::unordered_set<source_value_type, rxcpp::filtered_hash<source_value_type>>
remembered;

                                                distinct_observer(dest_type d)
                                                        : dest(d)
                                                {
                                                }
                                                void on_next(source_value_type v) const {
                                                        if (remembered.empty() || remembered.count(v) == 0) {
                                                                remembered.insert(v);
                                                                dest.on_next(v);
                                                        }
                                                }
                                                void on_error(std::exception_ptr e) const {
                                                        dest.on_error(e);
                                                }
                                                void on_completed() const {
                                                        dest.on_completed();
                                                }

                                                static subscriber<value_type, observer<value_type, this_type>> make(dest_type d) {
                                                        return make_subscriber<value_type>(d, this_type(d));
                                                }
                                        };

                                        template<class Subscriber>
                                        auto operator()(Subscriber dest) const
                                                -> decltype(distinct_observer<Subscriber>::make(std::move(dest))) {
                                                return      distinct_observer<Subscriber>::make(std::move(dest));
                                        }
                                };

                        }

                        /*! @copydoc rx-distinct.hpp
                        */
                        template<class... AN>
                        auto distinct(AN&&... an)
                                ->      operator_factory<distinct_tag, AN...> {
                                return operator_factory<distinct_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
                        }

                }

                template<>
                struct member_overload<distinct_tag>
                {
                        template<class Observable,
                        class SourceValue = rxu::value_type_t<Observable>,
                        class Enabled = rxu::enable_if_all_true_type_t<
                                is_observable<Observable>,
                                is_hashable<SourceValue >> ,
                        class Distinct = rxo::detail::distinct<SourceValue >>
                                static auto member(Observable&& o)
                                -> decltype(o.template lift<SourceValue>(Distinct())) {
                                        return  o.template lift<SourceValue>(Distinct());
                                }

                        template<class... AN>
                        static operators::detail::distinct_invalid_t<AN...> member(AN...) {
                                std::terminate();
                                return{};
                                static_assert(sizeof...(AN) == 10000, "distinct takes no arguments");
                        }
                };

}

#endif


/*! \file rx-distinct_until_changed.hpp

\brief For each item from this observable, filter out consequentially repeated values and emit only changes from the new observable that is
returned.

\return  Observable that emits those items from the source observable that are distinct from their immediate predecessors.

\sample
\snippet distinct_until_changed.cpp distinct_until_changed sample
\snippet output.txt distinct_until_changed sample
*/
```

```
#if !defined(RXCPP_OPERATORS_RX_DISTINCT_UNTIL_CHANGED_HPP)
#define RXCPP_OPERATORS_RX_DISTINCT_UNTIL_CHANGED_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace operators {

		namespace detail {

			template<class... AN>
			struct distinct_until_changed_invalid_arguments {};

			template<class... AN>
			struct distinct_until_changed_invalid : public
rxo::operator_base<distinct_until_changed_invalid_arguments<AN...>> {
				using type = observable<distinct_until_changed_invalid_arguments<AN...>,
distinct_until_changed_invalid<AN...>>;
			};
			template<class... AN>
			using distinct_until_changed_invalid_t = typename distinct_until_changed_invalid<AN...>::type;

			template<class T>
			struct distinct_until_changed
			{
				typedef rxu::decay_t<T> source_value_type;

				template<class Subscriber>
				struct distinct_until_changed_observer
				{
					typedef distinct_until_changed_observer<Subscriber> this_type;
					typedef source_value_type value_type;
					typedef rxu::decay_t<Subscriber> dest_type;
					typedef observer<value_type, this_type> observer_type;
					dest_type dest;
					mutable rxu::detail::maybe<source_value_type> remembered;

					distinct_until_changed_observer(dest_type d)
						: dest(d)
					{
					}
					void on_next(source_value_type v) const {
						if (remembered.empty() || v != remembered.get()) {
							remembered.reset(v);
							dest.on_next(v);
						}
					}
					void on_error(std::exception_ptr e) const {
						dest.on_error(e);
					}
					void on_completed() const {
						dest.on_completed();
					}

					static subscriber<value_type, observer_type> make(dest_type d) {
						return make_subscriber<value_type>(d, this_type(d));
					}
				};

				template<class Subscriber>
				auto operator()(Subscriber dest) const
					-> decltype(distinct_until_changed_observer<Subscriber>::make(std::move(dest))) {
					return	distinct_until_changed_observer<Subscriber>::make(std::move(dest));
				}
			};

		}

		/*! @copydoc rx-distinct_until_changed.hpp
		*/
		template<class... AN>
		auto distinct_until_changed(AN&&... an)
			->	operator_factory<distinct_until_changed_tag, AN...> {
			return operator_factory<distinct_until_changed_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
		}

	}

	template<>
	struct member_overload<distinct_until_changed_tag>
```

```
        {
                template<class Observable,
                class SourceValue = rxu::value_type_t<Observable>,
                class Enabled = rxu::enable_if_all_true_type_t<
                        is_observable<Observable>,
                        is_hashable<SourceValue >> ,
                class DistinctUntilChanged = rxo::detail::distinct_until_changed<SourceValue >>
                        static auto member(Observable&& o)
                        -> decltype(o.template lift<SourceValue>(DistinctUntilChanged())) {
                                return  o.template lift<SourceValue>(DistinctUntilChanged());
                        }

                template<class... AN>
                static operators::detail::distinct_until_changed_invalid_t<AN...> member(AN...) {
                        std::terminate();
                        return{};
                        static_assert(sizeof...(AN) == 10000, "distinct_until_changed takes no arguments");
                }
        };

}

#endif


/*! \file rx-element_at.hpp

\brief Pulls an item located at a specified index location in the sequence of items and emits that item as its own sole emission.

\param index  the index of the element to return.

\return An observable that emit an item located at a specified index location.

\sample
\snippet element_at.cpp element_at sample
\snippet output.txt element_at sample
*/

#if !defined(RXCPP_OPERATORS_RX_ELEMENT_AT_HPP)
#define RXCPP_OPERATORS_RX_ELEMENT_AT_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class... AN>
                        struct element_at_invalid_arguments {};

                        template<class... AN>
                        struct element_at_invalid : public rxo::operator_base<element_at_invalid_arguments<AN...>> {
                                using type = observable<element_at_invalid_arguments<AN...>, element_at_invalid<AN...>>;
                        };
                        template<class... AN>
                        using element_at_invalid_t = typename element_at_invalid<AN...>::type;

                        template<class T>
                        struct element_at {
                                typedef rxu::decay_t<T> source_value_type;

                                struct element_at_values {
                                        element_at_values(int i)
                                        : index(i)
                                        {
                                        }
                                        int index;
                                };

                                element_at_values initial;

                                element_at(int i)
                                        : initial(i)
                                {
                                }

                                template<class Subscriber>
                                struct element_at_observer : public element_at_values
                                {
                                        typedef element_at_observer<Subscriber> this_type;
```

```
                                        typedef source_value_type value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<value_type, this_type> observer_type;
                                        dest_type dest;
                                        mutable int current;

                                        element_at_observer(dest_type d, element_at_values v)
                                                : element_at_values(v),
                                                dest(d),
                                                current(0)
                                        {
                                        }
                                        void on_next(source_value_type v) const {
                                                if (current++ == this->index) {
                                                        dest.on_next(v);
                                                        dest.on_completed();
                                                }
                                        }
                                        void on_error(std::exception_ptr e) const {
                                                dest.on_error(e);
                                        }
                                        void on_completed() const {
                                                if (current <= this->index) {
                                                        dest.on_error(std::make_exception_ptr(std::range_error("index is out
of bounds")));
                                                }
                                        }

                                        static subscriber<value_type, observer_type> make(dest_type d, element_at_values v) {
                                                return make_subscriber<value_type>(d, this_type(d, v));
                                        }
                                };

                                template<class Subscriber>
                                auto operator()(Subscriber dest) const
                                        -> decltype(element_at_observer<Subscriber>::make(std::move(dest), initial)) {
                                        return      element_at_observer<Subscriber>::make(std::move(dest), initial);
                                }
                        };

                }

                /*! @copydoc rx-element_at.hpp
                */
                template<class... AN>
                auto element_at(AN&&... an)
                        ->      operator_factory<element_at_tag, AN...> {
                        return operator_factory<element_at_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
                }

        }

        template<>
        struct member_overload<element_at_tag>
        {
                template<class Observable,
                class Enabled = rxu::enable_if_all_true_type_t<
                        is_observable<Observable>
                >,
                class SourceValue = rxu::value_type_t<Observable>,
                class element_at = rxo::detail::element_at<SourceValue >>
                        static auto member(Observable&& o, int index)
                        -> decltype(o.template lift<SourceValue>(element_at(index))) {
                                return      o.template lift<SourceValue>(element_at(index));
                        }

                template<class... AN>
                static operators::detail::element_at_invalid_t<AN...> member(const AN...) {
                        std::terminate();
                        return{};
                        static_assert(sizeof...(AN) == 10000, "element_at takes (required int)");
                }
        };

}

#endif


/*! \file rx-filter.hpp

\brief For each item from this observable use Predicate to select which items to emit from the new observable that is returned.
```

```
\tparam Predicate  the type of the filter function

\param p  the filter function

\return  Observable that emits only those items emitted by the source observable that the filter evaluates as true.

\sample
\snippet filter.cpp filter sample
\snippet output.txt filter sample
*/

#if !defined(RXCPP_OPERATORS_RX_FILTER_HPP)
#define RXCPP_OPERATORS_RX_FILTER_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace operators {

		namespace detail {

			template<class... AN>
			struct filter_invalid_arguments {};

			template<class... AN>
			struct filter_invalid : public rxo::operator_base<filter_invalid_arguments<AN...>> {
				using type = observable<filter_invalid_arguments<AN...>, filter_invalid<AN...>>;
			};
			template<class... AN>
			using filter_invalid_t = typename filter_invalid<AN...>::type;

			template<class T, class Predicate>
			struct filter
			{
				typedef rxu::decay_t<T> source_value_type;
				typedef rxu::decay_t<Predicate> test_type;
				test_type test;

				filter(test_type t)
					: test(std::move(t))
				{
				}

				template<class Subscriber>
				struct filter_observer
				{
					typedef filter_observer<Subscriber> this_type;
					typedef source_value_type value_type;
					typedef rxu::decay_t<Subscriber> dest_type;
					typedef observer<value_type, this_type> observer_type;
					dest_type dest;
					test_type test;

					filter_observer(dest_type d, test_type t)
						: dest(std::move(d))
						, test(std::move(t))
					{
					}
					void on_next(source_value_type v) const {
						auto filtered = on_exception([&](){
							return !this->test(v); },
								dest);
							if (filtered.empty()) {
								return;
							}
							if (!filtered.get()) {
								dest.on_next(v);
							}
					}
					void on_error(std::exception_ptr e) const {
						dest.on_error(e);
					}
					void on_completed() const {
						dest.on_completed();
					}

					static subscriber<value_type, observer_type> make(dest_type d, test_type t) {
						return make_subscriber<value_type>(d, this_type(d, std::move(t)));
					}
				};
```

```
                                        template<class Subscriber>
                                        auto operator()(Subscriber dest) const
                                                -> decltype(filter_observer<Subscriber>::make(std::move(dest), test)) {
                                                return      filter_observer<Subscriber>::make(std::move(dest), test);
                                        }
                                };

                }

                /*! @copydoc rx-filter.hpp
                */
                template<class... AN>
                auto filter(AN&&... an)
                        ->      operator_factory<filter_tag, AN...> {
                        return operator_factory<filter_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
                }

        }

        template<>
        struct member_overload<filter_tag>
        {
                template<class Observable, class Predicate,
                class SourceValue = rxu::value_type_t<Observable>,
                class Filter = rxo::detail::filter<SourceValue, rxu::decay_t<Predicate>>>
                        static auto member(Observable&& o, Predicate&& p)
                        -> decltype(o.template lift<SourceValue>(Filter(std::forward<Predicate>(p)))) {
                                return      o.template lift<SourceValue>(Filter(std::forward<Predicate>(p)));
                        }

                template<class... AN>
                static operators::detail::filter_invalid_t<AN...> member(const AN&...) {
                        std::terminate();
                        return{};
                        static_assert(sizeof...(AN) == 10000, "filter takes (Predicate)");
                }
        };

}

#endif


/*! \file rx-group_by.hpp

\brief Return an observable that emits grouped_observables, each of which corresponds to a unique key value and each of which emits those items from the source observable that share that key value.

\tparam KeySelector      the type of the key extracting function
\tparam MarbleSelector   the type of the element extracting function
\tparam BinaryPredicate  the type of the key comparing function

\param  ks  a function that extracts the key for each item (optional)
\param  ms  a function that extracts the return element for each item (optional)
\param  p   a function that implements comparison of two keys (optional)

\return  Observable that emits values of grouped_observable type, each of which corresponds to a unique key value and each of which emits those items from the source observable that share that key value.

\sample
\snippet group_by.cpp group_by full intro
\snippet group_by.cpp group_by full sample
\snippet output.txt group_by full sample

\sample
\snippet group_by.cpp group_by sample
\snippet output.txt group_by sample
*/

#if !defined(RXCPP_OPERATORS_RX_GROUP_BY_HPP)
#define RXCPP_OPERATORS_RX_GROUP_BY_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class... AN>
```

```
struct group_by_invalid_arguments {};

template<class... AN>
struct group_by_invalid : public rxo::operator_base<group_by_invalid_arguments<AN...>> {
	using type = observable<group_by_invalid_arguments<AN...>, group_by_invalid<AN...>>;
};
template<class... AN>
using group_by_invalid_t = typename group_by_invalid<AN...>::type;

template<class T, class Selector>
struct is_group_by_selector_for {

	typedef rxu::decay_t<Selector> selector_type;
	typedef T source_value_type;

	struct tag_not_valid {};
	template<class CV, class CS>
	static auto check(int) -> decltype((*(CS*)nullptr)(*(CV*)nullptr));
	template<class CV, class CS>
	static tag_not_valid check(...);

	typedef decltype(check<source_value_type, selector_type>(0)) type;
	static const bool value = !std::is_same<type, tag_not_valid>::value;
};

template<class T, class Observable, class KeySelector, class MarbleSelector, class BinaryPredicate>
struct group_by_traits
{
	typedef T source_value_type;
	typedef rxu::decay_t<Observable> source_type;
	typedef rxu::decay_t<KeySelector> key_selector_type;
	typedef rxu::decay_t<MarbleSelector> marble_selector_type;
	typedef rxu::decay_t<BinaryPredicate> predicate_type;

	static_assert(is_group_by_selector_for<source_value_type, key_selector_type>::value, "group_by
KeySelector must be a function with the signature key_type(source_value_type)");

	typedef typename is_group_by_selector_for<source_value_type, key_selector_type>::type key_type;

	static_assert(is_group_by_selector_for<source_value_type, marble_selector_type>::value, "group_by
MarbleSelector must be a function with the signature marble_type(source_value_type)");

	typedef typename is_group_by_selector_for<source_value_type, marble_selector_type>::type
marble_type;

	typedef rxsub::subject<marble_type> subject_type;

	typedef std::map<key_type, typename subject_type::subscriber_type, predicate_type>
key_subscriber_map_type;

	typedef grouped_observable<key_type, marble_type> grouped_observable_type;
};

template<class T, class Observable, class KeySelector, class MarbleSelector, class BinaryPredicate>
struct group_by
{
	typedef group_by_traits<T, Observable, KeySelector, MarbleSelector, BinaryPredicate> traits_type;
	typedef typename traits_type::key_selector_type key_selector_type;
	typedef typename traits_type::marble_selector_type marble_selector_type;
	typedef typename traits_type::marble_type marble_type;
	typedef typename traits_type::predicate_type predicate_type;
	typedef typename traits_type::subject_type subject_type;
	typedef typename traits_type::key_type key_type;

	typedef typename traits_type::key_subscriber_map_type group_map_type;
	typedef std::vector<typename composite_subscription::weak_subscription> bindings_type;

	struct group_by_state_type
	{
		group_by_state_type(composite_subscription sl, predicate_type p)
		: source_lifetime(sl)
		, groups(p)
		, observers(0)
		{}
		composite_subscription source_lifetime;
		rxsc::worker worker;
		group_map_type groups;
		std::atomic<int> observers;
	};

	template<class Subscriber>
	static void stopsource(Subscriber&& dest, std::shared_ptr<group_by_state_type>& state) {
```

```
                ++state->observers;
                dest.add([state](){
                        if (!state->source_lifetime.is_subscribed()) {
                                return;
                        }
                        --state->observers;
                        if (state->observers == 0) {
                                state->source_lifetime.unsubscribe();
                        }
                });
        }

        struct group_by_values
        {
                group_by_values(key_selector_type ks, marble_selector_type ms, predicate_type p)
                : keySelector(std::move(ks))
                , marbleSelector(std::move(ms))
                , predicate(std::move(p))
                {
                }
                mutable key_selector_type keySelector;
                mutable marble_selector_type marbleSelector;
                mutable predicate_type predicate;
        };

        group_by_values initial;

        group_by(key_selector_type ks, marble_selector_type ms, predicate_type p)
                : initial(std::move(ks), std::move(ms), std::move(p))
        {
        }

        struct group_by_observable : public rxs::source_base<marble_type>
        {
                mutable std::shared_ptr<group_by_state_type> state;
                subject_type subject;
                key_type key;

                group_by_observable(std::shared_ptr<group_by_state_type> st, subject_type s, key_type
k)
                        : state(std::move(st))
                        , subject(std::move(s))
                        , key(k)
                {
                }

                template<class Subscriber>
                void on_subscribe(Subscriber&& o) const {
                        group_by::stopsource(o, state);
                        subject.get_observable().subscribe(std::forward<Subscriber>(o));
                }

                key_type on_get_key() {
                        return key;
                }
        };

        template<class Subscriber>
        struct group_by_observer : public group_by_values
        {
                typedef group_by_observer<Subscriber> this_type;
                typedef typename traits_type::grouped_observable_type value_type;
                typedef rxu::decay_t<Subscriber> dest_type;
                typedef observer<T, this_type> observer_type;

                dest_type dest;

                mutable std::shared_ptr<group_by_state_type> state;

                group_by_observer(composite_subscription l, dest_type d, group_by_values v)
                        : group_by_values(v)
                        , dest(std::move(d))
                        , state(std::make_shared<group_by_state_type>(l, group_by_values::predicate))
                {
                        group_by::stopsource(dest, state);
                }
                void on_next(T v) const {
                        auto selectedKey = on_exception(
                                [&](){
                                return this->keySelector(v); },
                                        [this](std::exception_ptr e){on_error(e); });
                                if (selectedKey.empty()) {
```

```
                                                                        return;
                                                                }
                                                                auto g = state->groups.find(selectedKey.get());
                                                                if (g == state->groups.end()) {
                                                                        if (!dest.is_subscribed()) {
                                                                                return;
                                                                        }
                                                                        auto sub = subject_type();
                                                                        g = state->groups.insert(std::make_pair(selectedKey.get(),
sub.get_subscriber())).first;

                dest.on_next(make_dynamic_grouped_observable<key_type, marble_type>(group_by_observable(state, sub, selectedKey.get())));
                                                                }
                                                                auto selectedMarble = on_exception(
                                                                        [&](){
                                                                        return this->marbleSelector(v); },
                                                                                [this](std::exception_ptr e){on_error(e); });
                                                                        if (selectedMarble.empty()) {
                                                                                return;
                                                                        }
                                                                        g->second.on_next(std::move(selectedMarble.get()));
                                                        }
                                                        void on_error(std::exception_ptr e) const {
                                                                for (auto& g : state->groups) {
                                                                        g.second.on_error(e);
                                                                }
                                                                dest.on_error(e);
                                                        }
                                                        void on_completed() const {
                                                                for (auto& g : state->groups) {
                                                                        g.second.on_completed();
                                                                }
                                                                dest.on_completed();
                                                        }

                                                        static subscriber<T, observer_type> make(dest_type d, group_by_values v) {
                                                                auto cs = composite_subscription();
                                                                return make_subscriber<T>(cs, observer_type(this_type(cs, std::move(d),
std::move(v))));
                                                        }
                                                };

                                                template<class Subscriber>
                                                auto operator()(Subscriber dest) const
                                                        -> decltype(group_by_observer<Subscriber>::make(std::move(dest), initial)) {
                                                        return      group_by_observer<Subscriber>::make(std::move(dest), initial);
                                                }
                                        };

                                        template<class KeySelector, class MarbleSelector, class BinaryPredicate>
                                        class group_by_factory
                                        {
                                                typedef rxu::decay_t<KeySelector> key_selector_type;
                                                typedef rxu::decay_t<MarbleSelector> marble_selector_type;
                                                typedef rxu::decay_t<BinaryPredicate> predicate_type;
                                                key_selector_type keySelector;
                                                marble_selector_type marbleSelector;
                                                predicate_type predicate;
                                        public:
                                                group_by_factory(key_selector_type ks, marble_selector_type ms, predicate_type p)
                                                        : keySelector(std::move(ks))
                                                        , marbleSelector(std::move(ms))
                                                        , predicate(std::move(p))
                                                {
                                                }
                                                template<class Observable>
                                                struct group_by_factory_traits
                                                {
                                                        typedef rxu::value_type_t<rxu::decay_t<Observable>> value_type;
                                                        typedef detail::group_by_traits<value_type, Observable, KeySelector, MarbleSelector,
BinaryPredicate> traits_type;
                                                        typedef detail::group_by<value_type, Observable, KeySelector, MarbleSelector,
BinaryPredicate> group_by_type;
                                                };
                                                template<class Observable>
                                                auto operator()(Observable&& source)
                                                        -> decltype(source.template lift<typename
group_by_factory_traits<Observable>::traits_type::grouped_observable_type>(typename
group_by_factory_traits<Observable>::group_by_type(std::move(keySelector), std::move(marbleSelector), std::move(predicate)))) {
                                                        return      source.template lift<typename
group_by_factory_traits<Observable>::traits_type::grouped_observable_type>(typename
group_by_factory_traits<Observable>::group_by_type(std::move(keySelector), std::move(marbleSelector), std::move(predicate)));
```

```
                                        }
                                };

                        }

                        /*! @copydoc rx-group_by.hpp
                        */
                        template<class... AN>
                        auto group_by(AN&&... an)
                                ->      operator_factory<group_by_tag, AN...> {
                                return operator_factory<group_by_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
                        }

        }

        template<>
        struct member_overload<group_by_tag>
        {
                template<class Observable, class KeySelector, class MarbleSelector, class BinaryPredicate,
                class SourceValue = rxu::value_type_t<Observable>,
                class Traits = rxo::detail::group_by_traits<SourceValue, rxu::decay_t<Observable>, KeySelector, MarbleSelector,
BinaryPredicate>,
                class GroupBy = rxo::detail::group_by<SourceValue, rxu::decay_t<Observable>, rxu::decay_t<KeySelector>,
rxu::decay_t<MarbleSelector>, rxu::decay_t<BinaryPredicate>>,
                class Value = typename Traits::grouped_observable_type>
                        static auto member(Observable&& o, KeySelector&& ks, MarbleSelector&& ms, BinaryPredicate&& p)
                        -> decltype(o.template lift<Value>(GroupBy(std::forward<KeySelector>(ks),
std::forward<MarbleSelector>(ms), std::forward<BinaryPredicate>(p)))) {
                                return      o.template lift<Value>(GroupBy(std::forward<KeySelector>(ks),
std::forward<MarbleSelector>(ms), std::forward<BinaryPredicate>(p)));
                        }

                template<class Observable, class KeySelector, class MarbleSelector,
                class BinaryPredicate = rxu::less,
                class SourceValue = rxu::value_type_t<Observable>,
                class Traits = rxo::detail::group_by_traits<SourceValue, rxu::decay_t<Observable>, KeySelector, MarbleSelector,
BinaryPredicate>,
                class GroupBy = rxo::detail::group_by<SourceValue, rxu::decay_t<Observable>, rxu::decay_t<KeySelector>,
rxu::decay_t<MarbleSelector>, rxu::decay_t<BinaryPredicate>>,
                class Value = typename Traits::grouped_observable_type>
                        static auto member(Observable&& o, KeySelector&& ks, MarbleSelector&& ms)
                        -> decltype(o.template lift<Value>(GroupBy(std::forward<KeySelector>(ks),
std::forward<MarbleSelector>(ms), rxu::less()))) {
                                return      o.template lift<Value>(GroupBy(std::forward<KeySelector>(ks),
std::forward<MarbleSelector>(ms), rxu::less()));
                        }


                template<class Observable, class KeySelector,
                class MarbleSelector = rxu::detail::take_at<0>,
                class BinaryPredicate = rxu::less,
                class SourceValue = rxu::value_type_t<Observable>,
                class Traits = rxo::detail::group_by_traits<SourceValue, rxu::decay_t<Observable>, KeySelector, MarbleSelector,
BinaryPredicate>,
                class GroupBy = rxo::detail::group_by<SourceValue, rxu::decay_t<Observable>, rxu::decay_t<KeySelector>,
rxu::decay_t<MarbleSelector>, rxu::decay_t<BinaryPredicate>>,
                class Value = typename Traits::grouped_observable_type>
                        static auto member(Observable&& o, KeySelector&& ks)
                        -> decltype(o.template lift<Value>(GroupBy(std::forward<KeySelector>(ks), rxu::detail::take_at<0>(),
rxu::less()))) {
                                return      o.template lift<Value>(GroupBy(std::forward<KeySelector>(ks), rxu::detail::take_at<0>(),
rxu::less()));
                        }

                template<class Observable,
                class KeySelector = rxu::detail::take_at<0>,
                class MarbleSelector = rxu::detail::take_at<0>,
                class BinaryPredicate = rxu::less,
                class Enabled = rxu::enable_if_all_true_type_t<
                        all_observables<Observable >> ,
                class SourceValue = rxu::value_type_t<Observable>,
                class Traits = rxo::detail::group_by_traits<SourceValue, rxu::decay_t<Observable>, KeySelector, MarbleSelector,
BinaryPredicate>,
                class GroupBy = rxo::detail::group_by<SourceValue, rxu::decay_t<Observable>, rxu::decay_t<KeySelector>,
rxu::decay_t<MarbleSelector>, rxu::decay_t<BinaryPredicate>>,
                class Value = typename Traits::grouped_observable_type>
                        static auto member(Observable&& o)
                        -> decltype(o.template lift<Value>(GroupBy(rxu::detail::take_at<0>(), rxu::detail::take_at<0>(), rxu::less()))) {
                                return      o.template lift<Value>(GroupBy(rxu::detail::take_at<0>(), rxu::detail::take_at<0>(),
rxu::less()));
                        }
```

```
                template<class... AN>
                static operators::detail::group_by_invalid_t<AN...> member(const AN&...) {
                        std::terminate();
                        return{};
                        static_assert(sizeof...(AN) == 10000, "group_by takes (optional KeySelector, optional MarbleSelector, optional
BinaryKeyPredicate), KeySelector takes (Observable::value_type) -> KeyValue, MarbleSelector takes (Observable::value_type) -> MarbleValue,
BinaryKeyPredicate takes (KeyValue, KeyValue) -> bool");
                }

        };

}

#endif



/*! \file rx-ignore_elements.hpp

\brief Do not emit any items from the source Observable, but allow termination notification (either onError or onCompleted) to pass through
unchanged.

\return Observable that emits termination notification from the source observable.

\sample
\snippet ignore_elements.cpp ignore_elements sample
\snippet output.txt ignore_elements sample
*/

#if !defined(RXCPP_OPERATORS_RX_IGNORE_ELEMENTS_HPP)
#define RXCPP_OPERATORS_RX_IGNORE_ELEMENTS_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class... AN>
                        struct ignore_elements_invalid_arguments {};

                        template<class... AN>
                        struct ignore_elements_invalid : public rxo::operator_base<ignore_elements_invalid_arguments<AN...>> {
                                using type = observable<ignore_elements_invalid_arguments<AN...>,
ignore_elements_invalid<AN...>>;
                        };
                        template<class... AN>
                        using ignore_elements_invalid_t = typename ignore_elements_invalid<AN...>::type;

                        template<class T>
                        struct ignore_elements {
                                typedef rxu::decay_t<T> source_value_type;

                                template<class Subscriber>
                                struct ignore_elements_observer
                                {
                                        typedef ignore_elements_observer<Subscriber> this_type;
                                        typedef source_value_type value_type;
                                        typedef rxu::decay_t<Subscriber> dest_type;
                                        typedef observer<value_type, this_type> observer_type;
                                        dest_type dest;

                                        ignore_elements_observer(dest_type d)
                                                : dest(d)
                                        {
                                        }

                                        void on_next(source_value_type) const {
                                                // no-op; ignore element
                                        }

                                        void on_error(std::exception_ptr e) const {
                                                dest.on_error(e);
                                        }

                                        void on_completed() const {
                                                dest.on_completed();
                                        }

                                        static subscriber<value_type, observer_type> make(dest_type d) {
```

```
                                                return make_subscriber<value_type>(d, this_type(d));
                                        }
                                };

                                template<class Subscriber>
                                auto operator()(Subscriber dest) const
                                        -> decltype(ignore_elements_observer<Subscriber>::make(std::move(dest))) {
                                        return      ignore_elements_observer<Subscriber>::make(std::move(dest));
                                }
                        };

                }

                /*! @copydoc rx-ignore_elements.hpp
                */
                template<class... AN>
                auto ignore_elements(AN&&... an)
                        ->     operator_factory<ignore_elements_tag, AN...> {
                        return operator_factory<ignore_elements_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
                }

        }

        template<>
        struct member_overload<ignore_elements_tag>
        {
                template<class Observable,
                class SourceValue = rxu::value_type_t<Observable>,
                class Enabled = rxu::enable_if_all_true_type_t<
                        is_observable<Observable >> ,
                class IgnoreElements = rxo::detail::ignore_elements<SourceValue >>
                        static auto member(Observable&& o)
                        -> decltype(o.template lift<SourceValue>(IgnoreElements())) {
                                return  o.template lift<SourceValue>(IgnoreElements());
                        }

                template<class... AN>
                static operators::detail::ignore_elements_invalid_t<AN...> member(AN...) {
                        std::terminate();
                        return{};
                        static_assert(sizeof...(AN) == 10000, "ignore_elements takes no arguments");
                }
        };

}

#endif


/*! \file rx-reduce.hpp

\brief For each item from this observable use Accumulator to combine items, when completed use ResultSelector to produce a value that will be emitted from the new observable that is returned.

\tparam Seed           the type of the initial value for the accumulator
\tparam Accumulator    the type of the data accumulating function
\tparam ResultSelector  the type of the result producing function

\param seed  the initial value for the accumulator
\param a     an accumulator function to be invoked on each item emitted by the source observable, the result of which will be used in the next accumulator call
\param rs    a result producing function that makes the final value from the last accumulator call result

\return  An observable that emits a single item that is the result of accumulating the output from the items emitted by the source observable.

Some basic reduce-type operators have already been implemented:
- rxcpp::operators::first
- rxcpp::operators::last
- rxcpp::operators::count
- rxcpp::operators::sum
- rxcpp::operators::average
- rxcpp::operators::min
- rxcpp::operators::max

\sample
Geometric mean of source values:
\snippet reduce.cpp reduce sample
\snippet output.txt reduce sample

If the source observable completes without emitting any items, the resulting observable emits the result of passing the initial seed to the result selector:
\snippet reduce.cpp reduce empty sample
```

```
\snippet output.txt reduce empty sample

If the accumulator raises an exception, it is returned by the resulting observable in on_error:
\snippet reduce.cpp reduce exception from accumulator sample
\snippet output.txt reduce exception from accumulator sample

The same for exceptions raised by the result selector:
\snippet reduce.cpp reduce exception from result selector sample
\snippet output.txt reduce exception from result selector sample
*/

#if !defined(RXCPP_OPERATORS_RX_REDUCE_HPP)
#define RXCPP_OPERATORS_RX_REDUCE_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

	namespace operators {

		namespace detail {

			template<class... AN>
			struct reduce_invalid_arguments {};

			template<class... AN>
			struct reduce_invalid : public rxo::operator_base<reduce_invalid_arguments<AN...>> {
				using type = observable<reduce_invalid_arguments<AN...>, reduce_invalid<AN...>>;
			};
			template<class... AN>
			using reduce_invalid_t = typename reduce_invalid<AN...>::type;

			template<class T, class Seed, class Accumulator>
			struct is_accumulate_function_for {

				typedef rxu::decay_t<Accumulator> accumulator_type;
				typedef rxu::decay_t<Seed> seed_type;
				typedef T source_value_type;

				struct tag_not_valid {};
				template<class CS, class CV, class CRS>
				static auto check(int) -> decltype((*(CRS*)nullptr)(*(CS*)nullptr, *(CV*)nullptr));
				template<class CS, class CV, class CRS>
				static tag_not_valid check(...);

				typedef decltype(check<seed_type, source_value_type, accumulator_type>(0)) type;
				static const bool value = std::is_same<type, seed_type>::value;
			};

			template<class Seed, class ResultSelector>
			struct is_result_function_for {

				typedef rxu::decay_t<ResultSelector> result_selector_type;
				typedef rxu::decay_t<Seed> seed_type;

				struct tag_not_valid {};

				template<class CS, class CRS>
				static auto check(int) -> decltype((*(CRS*)nullptr)(*(CS*)nullptr));
				template<class CS, class CRS>
				static tag_not_valid check(...);

				typedef rxu::decay_t<decltype(check<seed_type, result_selector_type>(0))> type;
				static const bool value = !std::is_same<type, tag_not_valid>::value;
			};

			template<class T, class Observable, class Accumulator, class ResultSelector, class Seed>
			struct reduce_traits
			{
				typedef rxu::decay_t<Observable> source_type;
				typedef rxu::decay_t<Accumulator> accumulator_type;
				typedef rxu::decay_t<ResultSelector> result_selector_type;
				typedef rxu::decay_t<Seed> seed_type;

				typedef T source_value_type;

				typedef typename is_result_function_for<seed_type, result_selector_type>::type value_type;
			};

			template<class T, class Observable, class Accumulator, class ResultSelector, class Seed>
			struct reduce : public operator_base<rxu::value_type_t<reduce_traits<T, Observable, Accumulator,
ResultSelector, Seed>>>
```

```
{
	typedef reduce<T, Observable, Accumulator, ResultSelector, Seed> this_type;
	typedef reduce_traits<T, Observable, Accumulator, ResultSelector, Seed> traits;

	typedef typename traits::source_type source_type;
	typedef typename traits::accumulator_type accumulator_type;
	typedef typename traits::result_selector_type result_selector_type;
	typedef typename traits::seed_type seed_type;

	typedef typename traits::source_value_type source_value_type;
	typedef typename traits::value_type value_type;

	struct reduce_initial_type
	{
		~reduce_initial_type()
		{
		}
		reduce_initial_type(source_type o, accumulator_type a, result_selector_type rs, seed_type
s)
			: source(std::move(o))
			, accumulator(std::move(a))
			, result_selector(std::move(rs))
			, seed(std::move(s))
		{
		}
		source_type source;
		accumulator_type accumulator;
		result_selector_type result_selector;
		seed_type seed;

	private:
		reduce_initial_type& operator=(reduce_initial_type o) RXCPP_DELETE;
	};
	reduce_initial_type initial;

	~reduce()
	{
	}
	reduce(source_type o, accumulator_type a, result_selector_type rs, seed_type s)
		: initial(std::move(o), std::move(a), std::move(rs), std::move(s))
	{
	}
	template<class Subscriber>
	void on_subscribe(Subscriber o) const {
		struct reduce_state_type
			: public reduce_initial_type
			, public std::enable_shared_from_this<reduce_state_type>
		{
			reduce_state_type(reduce_initial_type i, Subscriber scrbr)
			: reduce_initial_type(i)
			, source(i.source)
			, current(reduce_initial_type::seed)
			, out(std::move(scrbr))
			{
			}
			source_type source;
			seed_type current;
			Subscriber out;

		private:
			reduce_state_type& operator=(reduce_state_type o) RXCPP_DELETE;
		};
		auto state = std::make_shared<reduce_state_type>(initial, std::move(o));
		state->source.subscribe(
			state->out,
			// on_next
			[state](T t) {
			seed_type next = state->accumulator(std::move(state->current), std::move(t));
			state->current = std::move(next);
		},
			// on_error
			[state](std::exception_ptr e) {
			state->out.on_error(e);
		},
			// on_completed
			[state]() {
			auto result = on_exception(
				[&](){return state->result_selector(std::move(state->current)); },
				state->out);
			if (result.empty()) {
				return;
			}
```

```
                                        state->out.on_next(std::move(result.get()));
                                        state->out.on_completed();
                                }
                                );
                        }
                private:
                        reduce& operator=(reduce o) RXCPP_DELETE;
                };

                template<class T>
                struct initialize_seeder {
                        typedef T seed_type;
                        static seed_type seed() {
                                return seed_type{};
                        }
                };

                template<class T>
                struct average {
                        struct seed_type
                        {
                                seed_type()
                                : value()
                                , count(0)
                                {
                                }
                                rxu::maybe<T> value;
                                int count;
                                rxu::detail::maybe<double> stage;
                        };
                        static seed_type seed() {
                                return seed_type{};
                        }
                        template<class U>
                        seed_type operator()(seed_type a, U&& v) {
                                if (a.count != 0 &&
                                        (a.count == std::numeric_limits<int>::max() ||
                                        ((v > 0) && (*(a.value) > (std::numeric_limits<T>::max() - v))) ||
                                        ((v < 0) && (*(a.value) < (std::numeric_limits<T>::min() - v))))) {
                                        // would overflow, calc existing and reset for next batch
                                        // this will add error to the final result, but the alternative
                                        // is to fail on overflow
                                        double avg = static_cast<double>(*(a.value)) / a.count;
                                        if (!a.stage.empty()) {
                                                a.stage.reset((*a.stage + avg) / 2);
                                        }
                                        else {
                                                a.stage.reset(avg);
                                        }
                                        a.value.reset(std::forward<U>(v));
                                        a.count = 1;
                                }
                                else if (a.value.empty()) {
                                        a.value.reset(std::forward<U>(v));
                                        a.count = 1;
                                }
                                else {
                                        *(a.value) += v;
                                        ++a.count;
                                }
                                return a;
                        }
                        double operator()(seed_type a) {
                                if (!a.value.empty()) {
                                        double avg = static_cast<double>(*(a.value)) / a.count;
                                        if (!a.stage.empty()) {
                                                avg = (*a.stage + avg) / 2;
                                        }
                                        return avg;
                                }
                                throw rxcpp::empty_error("average() requires a stream with at least one value");
                        }
                };

                template<class T>
                struct sum {
                        typedef rxu::maybe<T> seed_type;
                        static seed_type seed() {
                                return seed_type();
                        }
                        template<class U>
                        seed_type operator()(seed_type a, U&& v) const {
```

```
                        if (a.empty())
                                a.reset(std::forward<U>(v));
                        else
                                *a = *a + v;
                        return a;
                }
                T operator()(seed_type a) const {
                        if (a.empty())
                                throw rxcpp::empty_error("sum() requires a stream with at least one value");
                        return *a;
                }
        };

        template<class T>
        struct max {
                typedef rxu::maybe<T> seed_type;
                static seed_type seed() {
                        return seed_type();
                }
                template<class U>
                seed_type operator()(seed_type a, U&& v) {
                        if (a.empty() || *a < v)
                                a.reset(std::forward<U>(v));
                        return a;
                }
                T operator()(seed_type a) {
                        if (a.empty())
                                throw rxcpp::empty_error("max() requires a stream with at least one value");
                        return *a;
                }
        };

        template<class T>
        struct min {
                typedef rxu::maybe<T> seed_type;
                static seed_type seed() {
                        return seed_type();
                }
                template<class U>
                seed_type operator()(seed_type a, U&& v) {
                        if (a.empty() || v < *a)
                                a.reset(std::forward<U>(v));
                        return a;
                }
                T operator()(seed_type a) {
                        if (a.empty())
                                throw rxcpp::empty_error("min() requires a stream with at least one value");
                        return *a;
                }
        };

        template<class T>
        struct first {
                using seed_type = rxu::maybe<T>;
                static seed_type seed() {
                        return seed_type();
                }
                template<class U>
                seed_type operator()(seed_type a, U&& v) {
                        a.reset(std::forward<U>(v));
                        return a;
                }
                T operator()(seed_type a) {
                        if (a.empty()) {
                                throw rxcpp::empty_error("first() requires a stream with at least one value");
                        }
                        return *a;
                }
        };

        template<class T>
        struct last {
                using seed_type = rxu::maybe<T>;
                static seed_type seed() {
                        return seed_type();
                }
                template<class U>
                seed_type operator()(seed_type a, U&& v) {
                        a.reset(std::forward<U>(v));
                        return a;
                }
                T operator()(seed_type a) {
```

```
                                        if (a.empty()) {
                                                throw rxcpp::empty_error("last() requires a stream with at least one value");
                                        }
                                        return *a;
                                }
                        };

                }

                /*! @copydoc rx-reduce.hpp
                */
                template<class... AN>
                auto reduce(AN&&... an)
                        ->      operator_factory<reduce_tag, AN...> {
                        return operator_factory<reduce_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
                }

                /*! \brief For each item from this observable reduce it by sending only the first item.

                \return  An observable that emits only the very first item emitted by the source observable.

                \sample
                \snippet math.cpp first sample
                \snippet output.txt first sample

                When the source observable calls on_error:
                \snippet math.cpp first empty sample
                \snippet output.txt first empty sample
                */
                inline auto first()
                        ->      operator_factory<first_tag> {
                        return operator_factory<first_tag>(std::tuple<>{});
                }

                /*! \brief For each item from this observable reduce it by sending only the last item.

                \return  An observable that emits only the very last item emitted by the source observable.

                \sample
                \snippet math.cpp last sample
                \snippet output.txt last sample

                When the source observable calls on_error:
                \snippet math.cpp last empty sample
                \snippet output.txt last empty sample
                */
                inline auto last()
                        ->      operator_factory<last_tag> {
                        return operator_factory<last_tag>(std::tuple<>{});
                }

                /*! \brief For each item from this observable reduce it by incrementing a count.

                \return  An observable that emits a single item: the number of elements emitted by the source observable.

                \sample
                \snippet math.cpp count sample
                \snippet output.txt count sample

                When the source observable calls on_error:
                \snippet math.cpp count error sample
                \snippet output.txt count error sample
                */
                inline auto count()
                        ->      operator_factory<reduce_tag, int, rxu::count, rxu::detail::take_at<0>> {
                        return operator_factory<reduce_tag, int, rxu::count, rxu::detail::take_at<0>>(std::make_tuple(0, rxu::count(),
rxu::take_at<0>()));
                }

                /*! \brief For each item from this observable reduce it by adding to the previous values and then dividing by the number of
items at the end.

                \return  An observable that emits a single item: the average of elements emitted by the source observable.

                \sample
                \snippet math.cpp average sample
                \snippet output.txt average sample

                When the source observable completes without emitting any items:
                \snippet math.cpp average empty sample
                \snippet output.txt average empty sample
```

```
                When the source observable calls on_error:
                \snippet math.cpp average error sample
                \snippet output.txt average error sample
                */
                inline auto average()
                        ->    operator_factory<average_tag> {
                        return operator_factory<average_tag>(std::tuple<>{});
                }

                /*! \brief For each item from this observable reduce it by adding to the previous items.

                \return  An observable that emits a single item: the sum of elements emitted by the source observable.

                \sample
                \snippet math.cpp sum sample
                \snippet output.txt sum sample

                When the source observable completes without emitting any items:
                \snippet math.cpp sum empty sample
                \snippet output.txt sum empty sample

                When the source observable calls on_error:
                \snippet math.cpp sum error sample
                \snippet output.txt sum error sample
                */
                inline auto sum()
                        ->    operator_factory<sum_tag> {
                        return operator_factory<sum_tag>(std::tuple<>{});
                }

                /*! \brief For each item from this observable reduce it by taking the min value of the previous items.

                \return  An observable that emits a single item: the min of elements emitted by the source observable.

                \sample
                \snippet math.cpp min sample
                \snippet output.txt min sample

                When the source observable completes without emitting any items:
                \snippet math.cpp min empty sample
                \snippet output.txt min empty sample

                When the source observable calls on_error:
                \snippet math.cpp min error sample
                \snippet output.txt min error sample
                */
                inline auto min()
                        ->    operator_factory<min_tag> {
                        return operator_factory<min_tag>(std::tuple<>{});
                }

                /*! \brief For each item from this observable reduce it by taking the max value of the previous items.

                \return  An observable that emits a single item: the max of elements emitted by the source observable.

                \sample
                \snippet math.cpp max sample
                \snippet output.txt max sample

                When the source observable completes without emitting any items:
                \snippet math.cpp max empty sample
                \snippet output.txt max empty sample

                When the source observable calls on_error:
                \snippet math.cpp max error sample
                \snippet output.txt max error sample
                */
                inline auto max()
                        ->    operator_factory<max_tag> {
                        return operator_factory<max_tag>(std::tuple<>{});
                }

        }

        template<>
        struct member_overload<reduce_tag>
        {

                template<class Observable, class Seed, class Accumulator, class ResultSelector,
                class Reduce = rxo::detail::reduce<rxu::value_type_t<Observable>, rxu::decay_t<Observable>,
rxu::decay_t<Accumulator>, rxu::decay_t<ResultSelector>, rxu::decay_t<Seed>>,
                class Value = rxu::value_type_t<Reduce>,
```

```
                class Result = observable<Value, Reduce >>
                        static Result member(Observable&& o, Seed&& s, Accumulator&& a, ResultSelector&& r)
                {
                                return Result(Reduce(std::forward<Observable>(o), std::forward<Accumulator>(a),
std::forward<ResultSelector>(r), std::forward<Seed>(s)));
                        }

                template<class Observable, class Seed, class Accumulator,
                class ResultSelector = rxu::detail::take_at<0>,
                class Reduce = rxo::detail::reduce<rxu::value_type_t<Observable>, rxu::decay_t<Observable>,
rxu::decay_t<Accumulator>, rxu::decay_t<ResultSelector>, rxu::decay_t<Seed>>,
                class Value = rxu::value_type_t<Reduce>,
                class Result = observable<Value, Reduce >>
                        static Result member(Observable&& o, Seed&& s, Accumulator&& a)
                {
                                return Result(Reduce(std::forward<Observable>(o), std::forward<Accumulator>(a),
rxu::detail::take_at<0>(), std::forward<Seed>(s)));
                        }

                template<class... AN>
                static operators::detail::reduce_invalid_t<AN...> member(AN...) {
                        std::terminate();
                        return{};
                        static_assert(sizeof...(AN) == 10000, "reduce takes (Seed, Accumulator, optional ResultSelector), Accumulator
takes (Seed, Observable::value_type) -> Seed, ResultSelector takes (Observable::value_type) -> ResultValue");
                }
        };

        template<>
        struct member_overload<first_tag>
        {
                template<class Observable,
                class SValue = rxu::value_type_t<Observable>,
                class Operation = operators::detail::first<SValue>,
                class Seed = decltype(Operation::seed()),
                class Accumulator = Operation,
                class ResultSelector = Operation,
                class TakeOne = decltype(((rxu::decay_t<Observable>*)nullptr)->take(1)),
                class Reduce = rxo::detail::reduce<SValue, rxu::decay_t<TakeOne>, rxu::decay_t<Accumulator>,
rxu::decay_t<ResultSelector>, rxu::decay_t<Seed>>,
                class RValue = rxu::value_type_t<Reduce>,
                class Result = observable<RValue, Reduce >>
                        static Result member(Observable&& o)
                {
                                return Result(Reduce(o.take(1), Operation{}, Operation{}, Operation::seed()));
                        }

                template<class... AN>
                static operators::detail::reduce_invalid_t<AN...> member(AN...) {
                        std::terminate();
                        return{};
                        static_assert(sizeof...(AN) == 10000, "first does not support Observable::value_type");
                }
        };

        template<>
        struct member_overload<last_tag>
        {
                template<class Observable,
                class SValue = rxu::value_type_t<Observable>,
                class Operation = operators::detail::last<SValue>,
                class Seed = decltype(Operation::seed()),
                class Accumulator = Operation,
                class ResultSelector = Operation,
                class Reduce = rxo::detail::reduce<SValue, rxu::decay_t<Observable>, rxu::decay_t<Accumulator>,
rxu::decay_t<ResultSelector>, rxu::decay_t<Seed>>,
                class RValue = rxu::value_type_t<Reduce>,
                class Result = observable<RValue, Reduce >>
                        static Result member(Observable&& o)
                {
                                return Result(Reduce(std::forward<Observable>(o), Operation{}, Operation{}, Operation::seed()));
                        }

                template<class... AN>
                static operators::detail::reduce_invalid_t<AN...> member(AN...) {
                        std::terminate();
                        return{};
                        static_assert(sizeof...(AN) == 10000, "last does not support Observable::value_type");
                }
        };

        template<>
```

```
	struct member_overload<sum_tag>
	{
		template<class Observable,
		class SValue = rxu::value_type_t<Observable>,
		class Operation = operators::detail::sum<SValue>,
		class Seed = decltype(Operation::seed()),
		class Accumulator = Operation,
		class ResultSelector = Operation,
		class Reduce = rxo::detail::reduce<SValue, rxu::decay_t<Observable>, rxu::decay_t<Accumulator>,
rxu::decay_t<ResultSelector>, rxu::decay_t<Seed>>,
		class RValue = rxu::value_type_t<Reduce>,
		class Result = observable<RValue, Reduce >>
			static Result member(Observable&& o)
		{
				return Result(Reduce(std::forward<Observable>(o), Operation{}, Operation{}, Operation::seed()));
			}

		template<class... AN>
		static operators::detail::reduce_invalid_t<AN...> member(AN...) {
			std::terminate();
			return{};
			static_assert(sizeof...(AN) == 10000, "sum does not support Observable::value_type");
		}
	};

	template<>
	struct member_overload<average_tag>
	{
		template<class Observable,
		class SValue = rxu::value_type_t<Observable>,
		class Operation = operators::detail::average<SValue>,
		class Seed = decltype(Operation::seed()),
		class Accumulator = Operation,
		class ResultSelector = Operation,
		class Reduce = rxo::detail::reduce<SValue, rxu::decay_t<Observable>, rxu::decay_t<Accumulator>,
rxu::decay_t<ResultSelector>, rxu::decay_t<Seed>>,
		class RValue = rxu::value_type_t<Reduce>,
		class Result = observable<RValue, Reduce >>
			static Result member(Observable&& o)
		{
				return Result(Reduce(std::forward<Observable>(o), Operation{}, Operation{}, Operation::seed()));
			}

		template<class... AN>
		static operators::detail::reduce_invalid_t<AN...> member(AN...) {
			std::terminate();
			return{};
			static_assert(sizeof...(AN) == 10000, "average does not support Observable::value_type");
		}
	};

	template<>
	struct member_overload<max_tag>
	{
		template<class Observable,
		class SValue = rxu::value_type_t<Observable>,
		class Operation = operators::detail::max<SValue>,
		class Seed = decltype(Operation::seed()),
		class Accumulator = Operation,
		class ResultSelector = Operation,
		class Reduce = rxo::detail::reduce<SValue, rxu::decay_t<Observable>, rxu::decay_t<Accumulator>,
rxu::decay_t<ResultSelector>, rxu::decay_t<Seed>>,
		class RValue = rxu::value_type_t<Reduce>,
		class Result = observable<RValue, Reduce >>
			static Result member(Observable&& o)
		{
				return Result(Reduce(std::forward<Observable>(o), Operation{}, Operation{}, Operation::seed()));
			}

		template<class... AN>
		static operators::detail::reduce_invalid_t<AN...> member(AN...) {
			std::terminate();
			return{};
			static_assert(sizeof...(AN) == 10000, "max does not support Observable::value_type");
		}
	};

	template<>
	struct member_overload<min_tag>
	{
		template<class Observable,
		class SValue = rxu::value_type_t<Observable>,
```

```
                        class Operation = operators::detail::min<SValue>,
                        class Seed = decltype(Operation::seed()),
                        class Accumulator = Operation,
                        class ResultSelector = Operation,
                        class Reduce = rxo::detail::reduce<SValue, rxu::decay_t<Observable>, rxu::decay_t<Accumulator>,
rxu::decay_t<ResultSelector>, rxu::decay_t<Seed>>,
                        class RValue = rxu::value_type_t<Reduce>,
                        class Result = observable<RValue, Reduce >>
                                static Result member(Observable&& o)
                        {
                                        return Result(Reduce(std::forward<Observable>(o), Operation{}, Operation{}, Operation::seed()));
                                }

                        template<class... AN>
                        static operators::detail::reduce_invalid_t<AN...> member(AN...) {
                                std::terminate();
                                return{};
                                static_assert(sizeof...(AN) == 10000, "min does not support Observable::value_type");
                        }
        };

}

#endif


/*! \file rx-with_latest_from.hpp

\brief For each item from the first observable select the latest value from all the observables to emit from the new observable that is returned.

\tparam AN  types of scheduler (optional), aggregate function (optional), and source observables

\param  an  scheduler (optional), aggregation function (optional), and source observables

\return  Observable that emits items that are the result of combining the items emitted by the source observables.

If scheduler is omitted, identity_current_thread is used.

If aggregation function is omitted, the resulting observable returns tuples of emitted items.

\sample

Neither scheduler nor aggregation function are present:
\snippet with_latest_from.cpp with_latest_from sample
\snippet output.txt with_latest_from sample

Only scheduler is present:
\snippet with_latest_from.cpp Coordination with_latest_from sample
\snippet output.txt Coordination with_latest_from sample

Only aggregation function is present:
\snippet with_latest_from.cpp Selector with_latest_from sample
\snippet output.txt Selector with_latest_from sample

Both scheduler and aggregation function are present:
\snippet with_latest_from.cpp Coordination+Selector with_latest_from sample
\snippet output.txt Coordination+Selector with_latest_from sample
*/

#if !defined(RXCPP_OPERATORS_RX_WITH_LATEST_FROM_HPP)
#define RXCPP_OPERATORS_RX_WITH_LATEST_FROM_HPP

//_include "../rx-includes.hpp"

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class... AN>
                        struct with_latest_from_invalid_arguments {};

                        template<class... AN>
                        struct with_latest_from_invalid : public rxo::operator_base<with_latest_from_invalid_arguments<AN...>> {
                                using type = observable<with_latest_from_invalid_arguments<AN...>,
with_latest_from_invalid<AN...>>;
                        };
                        template<class... AN>
                        using with_latest_from_invalid_t = typename with_latest_from_invalid<AN...>::type;

                        template<class Selector, class... ObservableN>
```

```
struct is_with_latest_from_selector_check {
	typedef rxu::decay_t<Selector> selector_type;

	struct tag_not_valid;
	template<class CS, class... CON>
	static auto check(int) -> decltype((*(CS*)nullptr)((*(typename CON::value_type*)nullptr)...));
	template<class CS, class... CON>
	static tag_not_valid check(...);

	using type = decltype(check<selector_type, rxu::decay_t<ObservableN>...>(0));

	static const bool value = !std::is_same<type, tag_not_valid>::value;
};

template<class Selector, class... ObservableN>
struct invalid_with_latest_from_selector {
	static const bool value = false;
};

template<class Selector, class... ObservableN>
struct is_with_latest_from_selector : public std::conditional<
	is_with_latest_from_selector_check<Selector, ObservableN...>::value,
	is_with_latest_from_selector_check<Selector, ObservableN...>,
	invalid_with_latest_from_selector<Selector, ObservableN... >> ::type {
};

template<class Selector, class... ON>
using result_with_latest_from_selector_t = typename is_with_latest_from_selector<Selector, ON...>::type;

template<class Coordination, class Selector, class... ObservableN>
struct with_latest_from_traits {

	typedef std::tuple<ObservableN...> tuple_source_type;
	typedef std::tuple<rxu::detail::maybe<typename ObservableN::value_type>...>
tuple_source_value_type;

	typedef rxu::decay_t<Selector> selector_type;
	typedef rxu::decay_t<Coordination> coordination_type;

	typedef typename is_with_latest_from_selector<selector_type, ObservableN...>::type value_type;
};

template<class Coordination, class Selector, class... ObservableN>
struct with_latest_from : public operator_base<rxu::value_type_t<with_latest_from_traits<Coordination,
Selector, ObservableN...>>>
{
	typedef with_latest_from<Coordination, Selector, ObservableN...> this_type;

	typedef with_latest_from_traits<Coordination, Selector, ObservableN...> traits;

	typedef typename traits::tuple_source_type tuple_source_type;
	typedef typename traits::tuple_source_value_type tuple_source_value_type;

	typedef typename traits::selector_type selector_type;

	typedef typename traits::coordination_type coordination_type;
	typedef typename coordination_type::coordinator_type coordinator_type;

	struct values
	{
		values(tuple_source_type o, selector_type s, coordination_type sf)
		: source(std::move(o))
		, selector(std::move(s))
		, coordination(std::move(sf))
		{
		}
		tuple_source_type source;
		selector_type selector;
		coordination_type coordination;
	};
	values initial;

	with_latest_from(coordination_type sf, selector_type s, tuple_source_type ts)
		: initial(std::move(ts), std::move(s), std::move(sf))
	{
	}

	template<int Index, class State>
	void subscribe_one(std::shared_ptr<State> state) const {

		typedef typename std::tuple_element<Index, tuple_source_type>::type::value_type
source_value_type;
```

```
                composite_subscription innercs;

                // when the out observer is unsubscribed all the
                // inner subscriptions are unsubscribed as well
                state->out.add(innercs);

                auto source = on_exception(
                        [&](){return state->coordinator.in(std::get<Index>(state->source)); },
                        state->out);
                if (source.empty()) {
                        return;
                }

                // this subscribe does not share the observer subscription
                // so that when it is unsubscribed the observer can be called
                // until the inner subscriptions have finished
                auto sink = make_subscriber<source_value_type>(
                        state->out,
                        innercs,
                        // on_next
                        [state](source_value_type st) {
                        auto& value = std::get<Index>(state->latest);

                        if (value.empty()) {
                                ++state->valuesSet;
                        }

                        value.reset(st);

                        if (state->valuesSet == sizeof... (ObservableN) && Index == 0) {
                                auto values = rxu::surely(state->latest);
                                auto selectedResult = rxu::apply(values, state->selector);
                                state->out.on_next(selectedResult);
                        }
                },
                        // on_error
                        [state](std::exception_ptr e) {
                        state->out.on_error(e);
                },
                        // on_completed
                        [state]() {
                        if (--state->pendingCompletions == 0) {
                                state->out.on_completed();
                        }
                }
                );
                auto selectedSink = on_exception(
                        [&](){return state->coordinator.out(sink); },
                        state->out);
                if (selectedSink.empty()) {
                        return;
                }
                source->subscribe(std::move(selectedSink.get()));
        }
        template<class State, int... IndexN>
        void subscribe_all(std::shared_ptr<State> state, rxu::values<int, IndexN...>) const {
                bool subscribed[] = { (subscribe_one<IndexN>(state), true)... };
                subscribed[0] = (*subscribed); // silence warning
        }

        template<class Subscriber>
        void on_subscribe(Subscriber scbr) const {
                static_assert(is_subscriber<Subscriber>::value, "subscribe must be passed a subscriber");

                typedef Subscriber output_type;

                struct with_latest_from_state_type
                        : public std::enable_shared_from_this<with_latest_from_state_type>
                        , public values
                {
                        with_latest_from_state_type(values i, coordinator_type coor, output_type oarg)
                        : values(std::move(i))
                        , pendingCompletions(sizeof... (ObservableN))
                        , valuesSet(0)
                        , coordinator(std::move(coor))
                        , out(std::move(oarg))
                        {
                        }

                        // on_completed on the output must wait until all the
                        // subscriptions have received on_completed
```

```
                                                                mutable int pendingCompletions;
                                                                mutable int valuesSet;
                                                                mutable tuple_source_value_type latest;
                                                                coordinator_type coordinator;
                                                                output_type out;
                                                        };

                                                        auto coordinator = initial.coordination.create_coordinator(scbr.get_subscription());

                                                        // take a copy of the values for each subscription
                                                        auto state = std::make_shared<with_latest_from_state_type>(initial,
std::move(coordinator), std::move(scbr));

                                                        subscribe_all(state, typename rxu::values_from<int, sizeof...(ObservableN)>::type());
                                                }
                                        };

                        }

                        /*! @copydoc rx-with_latest_from.hpp
                        */
                        template<class... AN>
                        auto with_latest_from(AN&&... an)
                                ->      operator_factory<with_latest_from_tag, AN...> {
                                return operator_factory<with_latest_from_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
                        }

        }

        template<>
        struct member_overload<with_latest_from_tag>
        {
                template<class Observable, class... ObservableN,
                class Enabled = rxu::enable_if_all_true_type_t<
                        all_observables<Observable, ObservableN... >> ,
                class with_latest_from = rxo::detail::with_latest_from<identity_one_worker, rxu::detail::pack, rxu::decay_t<Observable>,
rxu::decay_t<ObservableN>...>,
                class Value = rxu::value_type_t<with_latest_from>,
                class Result = observable<Value, with_latest_from >>
                        static Result member(Observable&& o, ObservableN&&... on)
                {
                                return Result(with_latest_from(identity_current_thread(), rxu::pack(),
std::make_tuple(std::forward<Observable>(o), std::forward<ObservableN>(on)...)));
                        }

                template<class Observable, class Selector, class... ObservableN,
                class Enabled = rxu::enable_if_all_true_type_t<
                        operators::detail::is_with_latest_from_selector<Selector, Observable, ObservableN...>,
                        all_observables<Observable, ObservableN... >> ,
                class ResolvedSelector = rxu::decay_t<Selector>,
                class with_latest_from = rxo::detail::with_latest_from<identity_one_worker, ResolvedSelector, rxu::decay_t<Observable>,
rxu::decay_t<ObservableN>...>,
                class Value = rxu::value_type_t<with_latest_from>,
                class Result = observable<Value, with_latest_from >>
                        static Result member(Observable&& o, Selector&& s, ObservableN&&... on)
                {
                                return Result(with_latest_from(identity_current_thread(), std::forward<Selector>(s),
std::make_tuple(std::forward<Observable>(o), std::forward<ObservableN>(on)...)));
                        }

                template<class Coordination, class Observable, class... ObservableN,
                class Enabled = rxu::enable_if_all_true_type_t<
                        is_coordination<Coordination>,
                        all_observables<Observable, ObservableN... >> ,
                class with_latest_from = rxo::detail::with_latest_from<Coordination, rxu::detail::pack, rxu::decay_t<Observable>,
rxu::decay_t<ObservableN>...>,
                class Value = rxu::value_type_t<with_latest_from>,
                class Result = observable<Value, with_latest_from >>
                        static Result member(Observable&& o, Coordination&& cn, ObservableN&&... on)
                {
                                return Result(with_latest_from(std::forward<Coordination>(cn), rxu::pack(),
std::make_tuple(std::forward<Observable>(o), std::forward<ObservableN>(on)...)));
                        }

                template<class Coordination, class Selector, class Observable, class... ObservableN,
                class Enabled = rxu::enable_if_all_true_type_t<
                        is_coordination<Coordination>,
                        operators::detail::is_with_latest_from_selector<Selector, Observable, ObservableN...>,
                        all_observables<Observable, ObservableN... >> ,
                class ResolvedSelector = rxu::decay_t<Selector>,
                class with_latest_from = rxo::detail::with_latest_from<Coordination, ResolvedSelector, rxu::decay_t<Observable>,
rxu::decay_t<ObservableN>...>,
```

```
                        class Value = rxu::value_type_t<with_latest_from>,
                        class Result = observable<Value, with_latest_from >>
                                static Result member(Observable&& o, Coordination&& cn, Selector&& s, ObservableN&&... on)
                        {
                                        return Result(with_latest_from(std::forward<Coordination>(cn), std::forward<Selector>(s),
std::make_tuple(std::forward<Observable>(o), std::forward<ObservableN>(on)...)));
                                }

                        template<class... AN>
                        static operators::detail::with_latest_from_invalid_t<AN...> member(const AN&...) {
                                std::terminate();
                                return{};
                                static_assert(sizeof...(AN) == 10000, "with_latest_from takes (optional Coordination, optional Selector, required
Observable, optional Observable...), Selector takes (Observable::value_type...)");
                        }
        };

}

#endif


#if !defined(RXCPP_OPERATORS_RX_ZIP_HPP)
#define RXCPP_OPERATORS_RX_ZIP_HPP

//_include "../rx-includes.hpp"

/*! \file rx-zip.hpp

\brief Bring by one item from all given observables and select a value to emit from the new observable that is returned.

\tparam AN  types of scheduler (optional), aggregate function (optional), and source observables

\param  an  scheduler (optional), aggregation function (optional), and source observables

\return  Observable that emits the result of combining the items emitted and brought by one from each of the source observables.

If scheduler is omitted, identity_current_thread is used.

If aggregation function is omitted, the resulting observable returns tuples of emitted items.

\sample

Neither scheduler nor aggregation function are present:
\snippet zip.cpp zip sample
\snippet output.txt zip sample

Only scheduler is present:
\snippet zip.cpp Coordination zip sample
\snippet output.txt Coordination zip sample

Only aggregation function is present:
\snippet zip.cpp Selector zip sample
\snippet output.txt Selector zip sample

Both scheduler and aggregation function are present:
\snippet zip.cpp Coordination+Selector zip sample
\snippet output.txt Coordination+Selector zip sample
*/

namespace rxcpp {

        namespace operators {

                namespace detail {

                        template<class Observable>
                        struct zip_source_state
                        {
                                using value_type = rxu::value_type_t<Observable>;
                                zip_source_state()
                                        : completed(false)
                                {
                                }
                                std::list<value_type> values;
                                bool completed;
                        };

                        struct values_not_empty {
                                template<class Observable>
                                bool operator()(zip_source_state<Observable>& source) const {
                                        return !source.values.empty();
```

```
		}
};

struct source_completed_values_empty {
		template<class Observable>
		bool operator()(zip_source_state<Observable>& source) const {
			return source.completed && source.values.empty();
		}
};

struct extract_value_front {
		template<class Observable, class Value = rxu::value_type_t<Observable>>
		Value operator()(zip_source_state<Observable>& source) const {
			auto val = std::move(source.values.front());
			source.values.pop_front();
			return val;
		}
};

template<class... AN>
struct zip_invalid_arguments {};

template<class... AN>
struct zip_invalid : public rxo::operator_base<zip_invalid_arguments<AN...>> {
		using type = observable<zip_invalid_arguments<AN...>, zip_invalid<AN...>>;
};
template<class... AN>
using zip_invalid_t = typename zip_invalid<AN...>::type;

template<class Selector, class... ObservableN>
struct is_zip_selector_check {
		typedef rxu::decay_t<Selector> selector_type;

		struct tag_not_valid;
		template<class CS, class... CON>
		static auto check(int) -> decltype((*(CS*)nullptr)((*(typename CON::value_type*)nullptr)...));
		template<class CS, class... CON>
		static tag_not_valid check(...);

		using type = decltype(check<selector_type, rxu::decay_t<ObservableN>...>(0));

		static const bool value = !std::is_same<type, tag_not_valid>::value;
};

template<class Selector, class... ObservableN>
struct invalid_zip_selector {
		static const bool value = false;
};

template<class Selector, class... ObservableN>
struct is_zip_selector : public std::conditional<
		is_zip_selector_check<Selector, ObservableN...>::value,
		is_zip_selector_check<Selector, ObservableN...>,
		invalid_zip_selector<Selector, ObservableN... >> ::type {
};

template<class Selector, class... ON>
using result_zip_selector_t = typename is_zip_selector<Selector, ON...>::type;

template<class Coordination, class Selector, class... ObservableN>
struct zip_traits {
		typedef std::tuple<rxu::decay_t<ObservableN>...> tuple_source_type;
		typedef std::tuple<zip_source_state<ObservableN>...> tuple_source_values_type;

		typedef rxu::decay_t<Selector> selector_type;
		typedef rxu::decay_t<Coordination> coordination_type;

		typedef typename is_zip_selector<selector_type, ObservableN...>::type value_type;
};

template<class Coordination, class Selector, class... ObservableN>
struct zip : public operator_base<rxu::value_type_t<zip_traits<Coordination, Selector, ObservableN...>>>
{
		typedef zip<Coordination, Selector, ObservableN...> this_type;

		typedef zip_traits<Coordination, Selector, ObservableN...> traits;

		typedef typename traits::tuple_source_type tuple_source_type;
		typedef typename traits::tuple_source_values_type tuple_source_values_type;

		typedef typename traits::selector_type selector_type;
```

```
                                        typedef typename traits::coordination_type coordination_type;
                                        typedef typename coordination_type::coordinator_type coordinator_type;

                                        struct values
                                        {
                                                values(tuple_source_type o, selector_type s, coordination_type sf)
                                                : source(std::move(o))
                                                , selector(std::move(s))
                                                , coordination(std::move(sf))
                                                {
                                                }
                                                tuple_source_type source;
                                                selector_type selector;
                                                coordination_type coordination;
                                        };
                                        values initial;

                                        zip(coordination_type sf, selector_type s, tuple_source_type ts)
                                                : initial(std::move(ts), std::move(s), std::move(sf))
                                        {
                                        }

                                        template<int Index, class State>
                                        void subscribe_one(std::shared_ptr<State> state) const {

                                                typedef typename std::tuple_element<Index, tuple_source_type>::type::value_type
source_value_type;

                                                composite_subscription innercs;

                                                // when the out observer is unsubscribed all the
                                                // inner subscriptions are unsubscribed as well
                                                state->out.add(innercs);

                                                auto source = on_exception(
                                                        [&](){return state->coordinator.in(std::get<Index>(state->source)); },
                                                        state->out);
                                                if (source.empty()) {
                                                        return;
                                                }

                                                // this subscribe does not share the observer subscription
                                                // so that when it is unsubscribed the observer can be called
                                                // until the inner subscriptions have finished
                                                auto sink = make_subscriber<source_value_type>(
                                                        state->out,
                                                        innercs,
                                                        // on_next
                                                        [state](source_value_type st) {
                                                        auto& values = std::get<Index>(state->pending).values;
                                                        values.push_back(st);
                                                        if (rxu::apply_to_each(state->pending, values_not_empty(),
rxu::all_values_true())) {
                                                                auto selectedResult = rxu::apply_to_each(state->pending,
extract_value_front(), state->selector);
                                                                state->out.on_next(selectedResult);
                                                        }
                                                        if (rxu::apply_to_each(state->pending, source_completed_values_empty(),
rxu::any_value_true())) {
                                                                state->out.on_completed();
                                                        }
                                                },
                                                        // on_error
                                                        [state](std::exception_ptr e) {
                                                        state->out.on_error(e);
                                                },
                                                        // on_completed
                                                        [state]() {
                                                        auto& completed = std::get<Index>(state->pending).completed;
                                                        completed = true;
                                                        if (--state->pendingCompletions == 0) {
                                                                state->out.on_completed();
                                                        }
                                                }
                                                );
                                                auto selectedSink = on_exception(
                                                        [&](){return state->coordinator.out(sink); },
                                                        state->out);
                                                if (selectedSink.empty()) {
                                                        return;
                                                }
                                                source->subscribe(std::move(selectedSink.get()));
```

```
                                        }
                                        template<class State, int... IndexN>
                                        void subscribe_all(std::shared_ptr<State> state, rxu::values<int, IndexN...>) const {
                                                bool subscribed[] = { (subscribe_one<IndexN>(state), true)... };
                                                subscribed[0] = (*subscribed); // silence warning
                                        }

                                        template<class Subscriber>
                                        void on_subscribe(Subscriber scbr) const {
                                                static_assert(is_subscriber<Subscriber>::value, "subscribe must be passed a subscriber");

                                                typedef Subscriber output_type;

                                                struct zip_state_type
                                                        : public std::enable_shared_from_this<zip_state_type>
                                                        , public values
                                                {
                                                        zip_state_type(values i, coordinator_type coor, output_type oarg)
                                                        : values(std::move(i))
                                                        , pendingCompletions(sizeof... (ObservableN))
                                                        , valuesSet(0)
                                                        , coordinator(std::move(coor))
                                                        , out(std::move(oarg))
                                                        {
                                                        }

                                                        // on_completed on the output must wait until all the
                                                        // subscriptions have received on_completed
                                                        mutable int pendingCompletions;
                                                        mutable int valuesSet;
                                                        mutable tuple_source_values_type pending;
                                                        coordinator_type coordinator;
                                                        output_type out;
                                                };

                                                auto coordinator = initial.coordination.create_coordinator(scbr.get_subscription());

                                                // take a copy of the values for each subscription
                                                auto state = std::make_shared<zip_state_type>(initial, std::move(coordinator),
std::move(scbr));

                                                subscribe_all(state, typename rxu::values_from<int, sizeof...(ObservableN)>::type());
                                        }
                                };

                        }

                        /*! @copydoc rx-zip.hpp
                        */
                        template<class... AN>
                        auto zip(AN&&... an)
                                ->      operator_factory<zip_tag, AN...> {
                                return operator_factory<zip_tag, AN...>(std::make_tuple(std::forward<AN>(an)...));
                        }

                }

                template<>
                struct member_overload<zip_tag>
                {
                        template<class Observable, class... ObservableN,
                        class Enabled = rxu::enable_if_all_true_type_t<
                                all_observables<Observable, ObservableN... >> ,
                        class Zip = rxo::detail::zip<identity_one_worker, rxu::detail::pack, rxu::decay_t<Observable>,
rxu::decay_t<ObservableN>...>,
                        class Value = rxu::value_type_t<Zip>,
                        class Result = observable<Value, Zip >>
                                static Result member(Observable&& o, ObservableN&&... on)
                        {
                                        return Result(Zip(identity_current_thread(), rxu::pack(),
std::make_tuple(std::forward<Observable>(o), std::forward<ObservableN>(on)...)));
                                }

                        template<class Observable, class Selector, class... ObservableN,
                        class Enabled = rxu::enable_if_all_true_type_t<
                                operators::detail::is_zip_selector<Selector, Observable, ObservableN...>,
                                all_observables<Observable, ObservableN... >> ,
                        class ResolvedSelector = rxu::decay_t<Selector>,
                        class Zip = rxo::detail::zip<identity_one_worker, ResolvedSelector, rxu::decay_t<Observable>,
rxu::decay_t<ObservableN>...>,
                        class Value = rxu::value_type_t<Zip>,
                        class Result = observable<Value, Zip >>
```

```
                                        static Result member(Observable&& o, Selector&& s, ObservableN&&... on)
                        {
                                        return Result(Zip(identity_current_thread(), std::forward<Selector>(s),
std::make_tuple(std::forward<Observable>(o), std::forward<ObservableN>(on)...)));
                                        }

                        template<class Coordination, class Observable, class... ObservableN,
                        class Enabled = rxu::enable_if_all_true_type_t<
                                        is_coordination<Coordination>,
                                        all_observables<Observable, ObservableN... >> ,
                        class Zip = rxo::detail::zip<Coordination, rxu::detail::pack, rxu::decay_t<Observable>, rxu::decay_t<ObservableN>...>,
                        class Value = rxu::value_type_t<Zip>,
                        class Result = observable<Value, Zip >>
                                        static Result member(Observable&& o, Coordination&& cn, ObservableN&&... on)
                        {
                                        return Result(Zip(std::forward<Coordination>(cn), rxu::pack(),
std::make_tuple(std::forward<Observable>(o), std::forward<ObservableN>(on)...)));
                                        }

                        template<class Coordination, class Selector, class Observable, class... ObservableN,
                        class Enabled = rxu::enable_if_all_true_type_t<
                                        is_coordination<Coordination>,
                                        operators::detail::is_zip_selector<Selector, Observable, ObservableN...>,
                                        all_observables<Observable, ObservableN... >> ,
                        class ResolvedSelector = rxu::decay_t<Selector>,
                        class Zip = rxo::detail::zip<Coordination, ResolvedSelector, rxu::decay_t<Observable>, rxu::decay_t<ObservableN>...>,
                        class Value = rxu::value_type_t<Zip>,
                        class Result = observable<Value, Zip >>
                                        static Result member(Observable&& o, Coordination&& cn, Selector&& s, ObservableN&&... on)
                        {
                                        return Result(Zip(std::forward<Coordination>(cn), std::forward<Selector>(s),
std::make_tuple(std::forward<Observable>(o), std::forward<ObservableN>(on)...)));
                                        }

                        template<class... AN>
                        static operators::detail::zip_invalid_t<AN...> member(const AN&...) {
                                        std::terminate();
                                        return{};
                                        static_assert(sizeof...(AN) == 10000, "zip takes (optional Coordination, optional Selector, required Observable,
optional Observable...), Selector takes (Observable::value_type...)");
                        }
        };

}

#endif

#endif

#pragma pop_macro("min")
#pragma pop_macro("max")

#endif


#endif
```