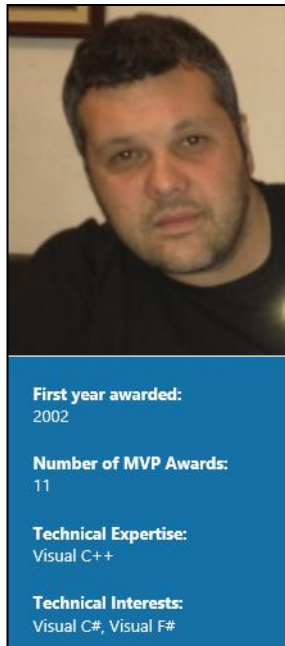
# Introdução a *Multi-threading* com C++ 11

(C++ 11 == ISO/IEC 14882:2011)


Fabio Galuppo, M.Sc.

http://fabiogaluppo.com

fabiogaluppo@acm.org

**First year awarded:**
2002

**Number of MVP Awards:**
11

**Technical Expertise:**
Visual C++

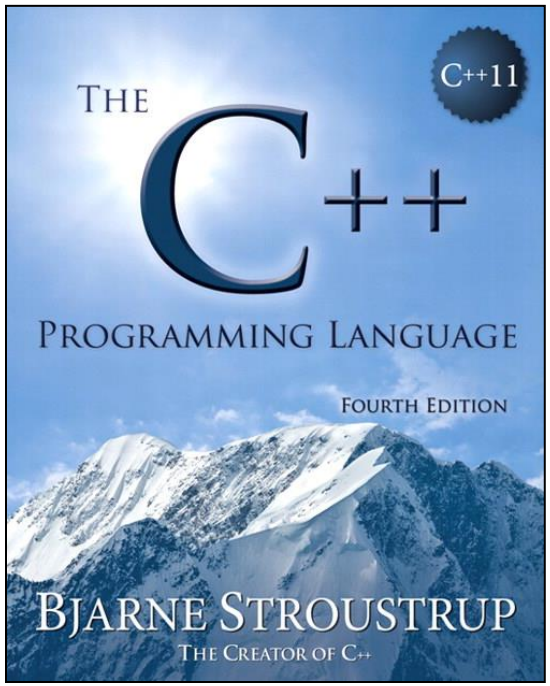
**Technical Interests:**
Visual C#, Visual F#

# The C++ Programming Language

C++ is a general purpose programming language with a bias towards systems programming that

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming.

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer.




Domain-specific abstraction

Fortran

Cobol

General-purpose abstraction

Simula

Java

C++

C++11

C#

Direct mapping to hardware

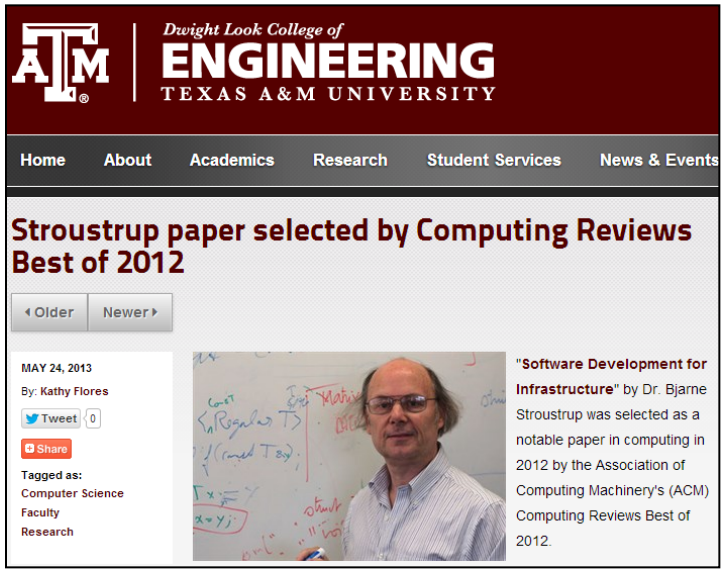
Assembler

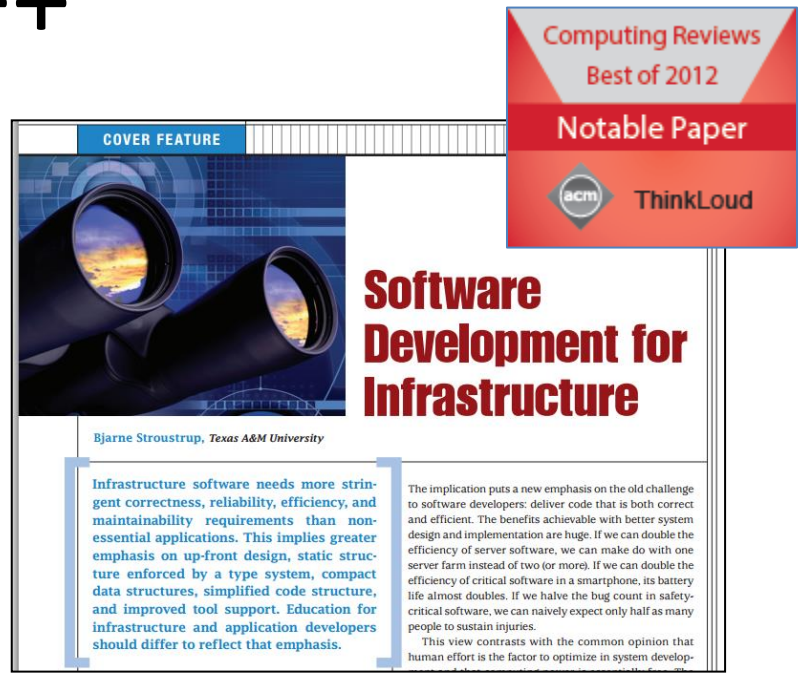
BCPL

C

Fonte: http://www.stroustrup.com/

# C++




IEEE Computer Magazine Volume:45, Issue: 1
http://dx.doi.org/10.1109/MC.2011.353

"A light-weight abstraction programming language

Key strengths:

- software infrastructure
- resource-constrained applications"

C++ 11
The Future is here

-- http://www.infoq.com/presentations/Cplusplus-11-Bjarne-Stroustrup

# Linguagem + Biblioteca

C++ 14: https://github.com/cplusplus/draft

| | |
|---|---|
| **Document Number:** | N3797 |
| **Date:** | 2013-10-13 |
| **Revises:** | N3691 |
| **Reply to:** | Stefanus Du Toit |
| | cxxeditor@gmail.com |

**Working Draft, Standard for Programming Language C++**

acesso em: 25 de Maio de 2014

C++ 11: ISO/IEC 14882:2011

**Abstract**

ISO/IEC 14882:2011 specifies requirements for implementations of the C++ programming language. The first such requirement is that they implement the language, and so ISO/IEC 14882:2011 also defines C++. Other requirements and relaxations of the first requirement appear at various places within ISO/IEC 14882:2011.

C++ is a general purpose programming language based on the C programming language as specified in ISO/IEC 9899:1999. In addition to the facilities provided by C, C++ provides additional data types, classes, templates, exceptions, namespaces, operator overloading, function name overloading, references, free store management operators, and additional library facilities.

Table 4 — Keywords

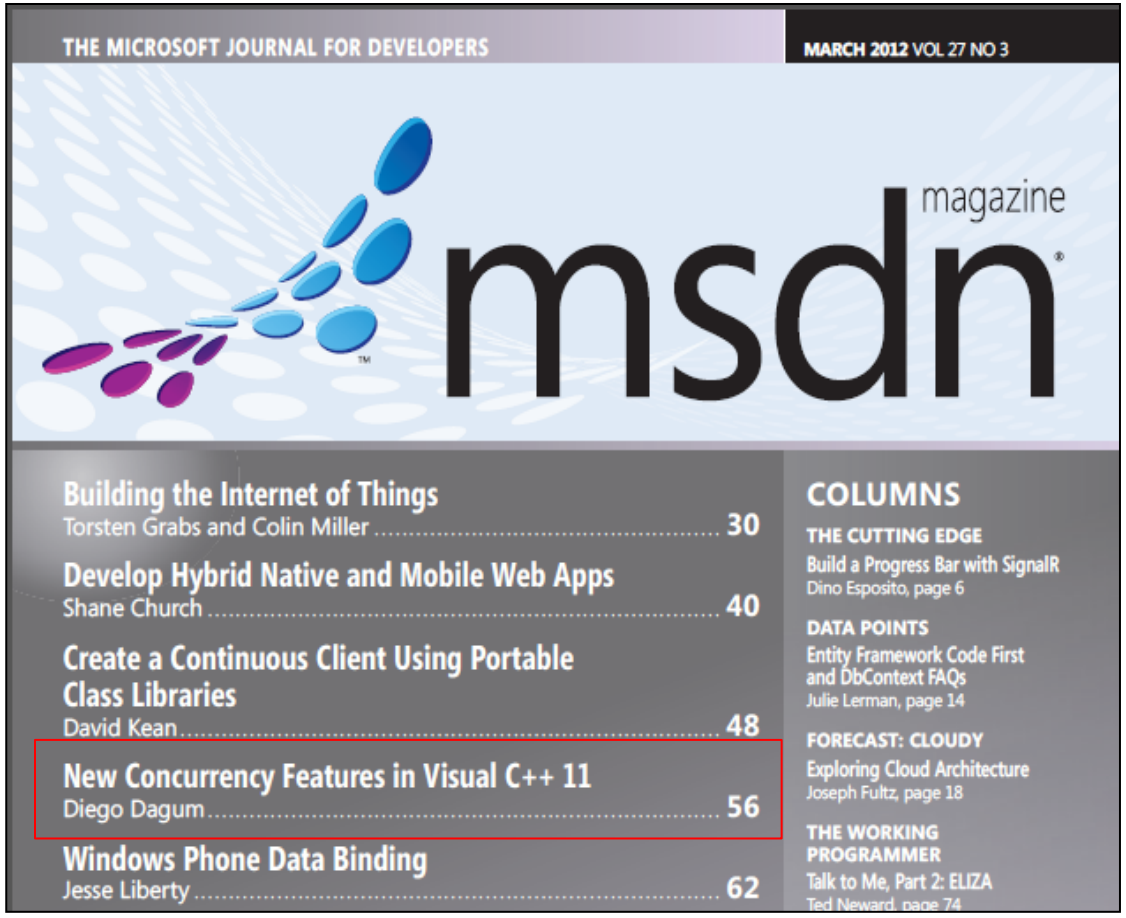
| | | | | |
|---|---|---|---|---|
| alignas | continue | friend | register | true |
| alignof | decltype | goto | reinterpret_cast | try |
| asm | default | if | return | typedef |
| auto | delete | inline | short | typeid |
| bool | do | int | signed | typename |
| break | double | long | sizeof | union |
| case | dynamic_cast | mutable | static | unsigned |
| catch | else | namespace | static_assert | using |
| char | enum | new | static_cast | virtual |
| char16_t | explicit | noexcept | struct | void |
| char32_t | export | nullptr | switch | volatile |
| class | extern | operator | template | wchar_t |
| const | false | private | this | while |
| constexpr | float | protected | thread_local | |
| const_cast | for | public | throw | |

2.12 Keywords [lex.key] [2. Lexical conventions]

# Compiladores


Summary of C++11 Feature Availability: http://www.aristeia.com/C++11/C++11FeatureAvailability.htm

# Programação Concorrente com C++ 11




artigo online: http://msdn.microsoft.com/en-us/magazine/hh852594.aspx
código fonte: http://archive.msdn.microsoft.com/mag201203CPP

This article discusses Visual C++ 11, a prerelease technology. All related information is subject to change.

This article discusses:

- Parallel execution
- Asynchronous tasks
- Threads
- Variables and exceptions
- Synchronization
- Atomic types
- Mutexes and locks
- Condition variables

Technologies discussed:

Visual C++ 11

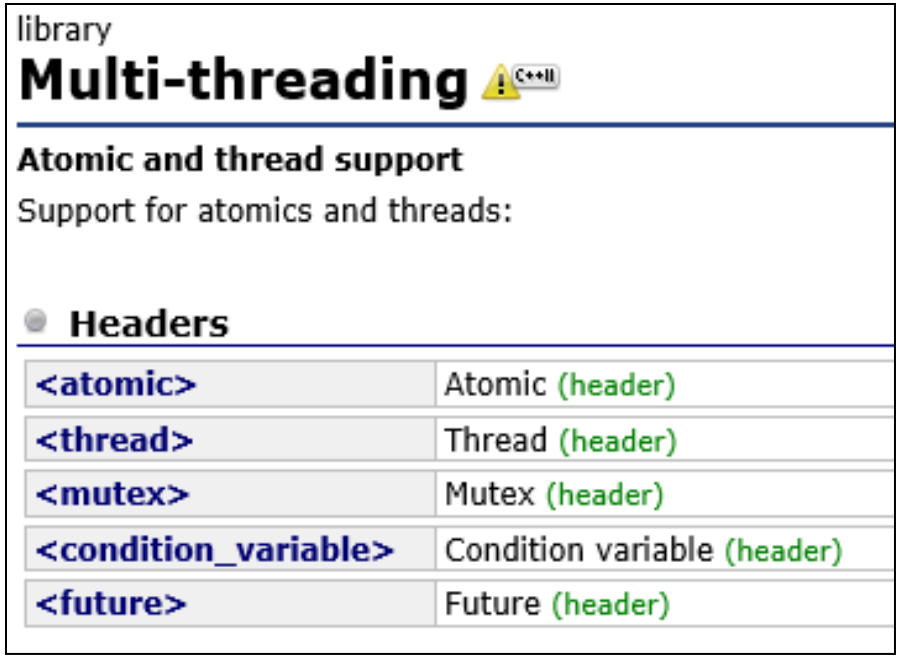
Code download available at:

code.msdn.microsoft.com/mag201203CPP

**DIEGO DAGUM** *is a software developer with more than 20 years of experience. He's currently a Visual C++ community program manager with Microsoft.*

**THANKS** to the following technical experts for reviewing this article: *David Cravey, Alon Fliess, Fabio Galuppo and Marc Gregoire*

# *Multi-threading with* **`std::`**



library

## Multi-threading

**Atomic and thread support**

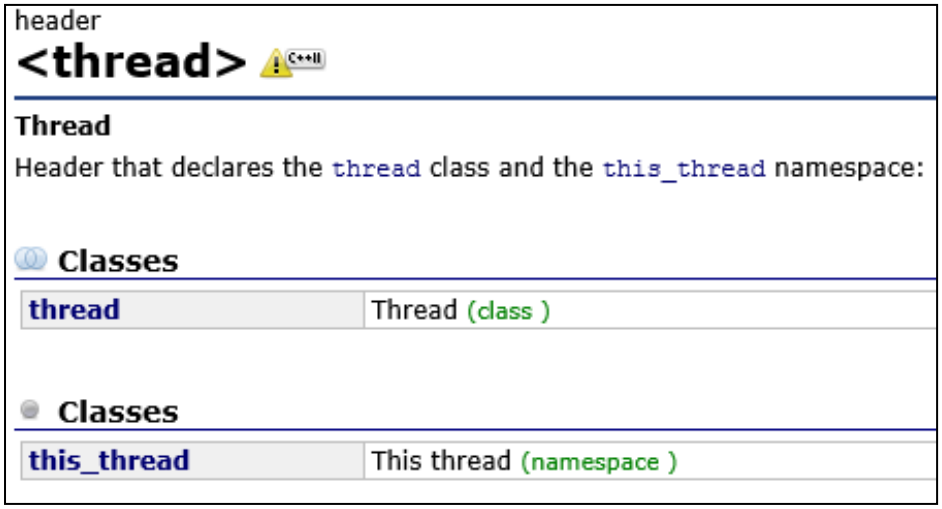
Support for atomics and threads:

### Headers

| | |
|---|---|
| **<atomic>** | Atomic (header) |
| **<thread>** | Thread (header) |
| **<mutex>** | Mutex (header) |
| **<condition_variable>** | Condition variable (header) |
| **<future>** | Future (header) |

http://www.cplusplus.com/reference/multithreading/

# #include <thread>



header
**<thread>**

**Thread**

Header that declares the `thread` class and the `this_thread` namespace:

**Classes**

| | |
|---|---|
| **thread** | Thread (class ) |

**Classes**

| | |
|---|---|
| **this_thread** | This thread (namespace ) |

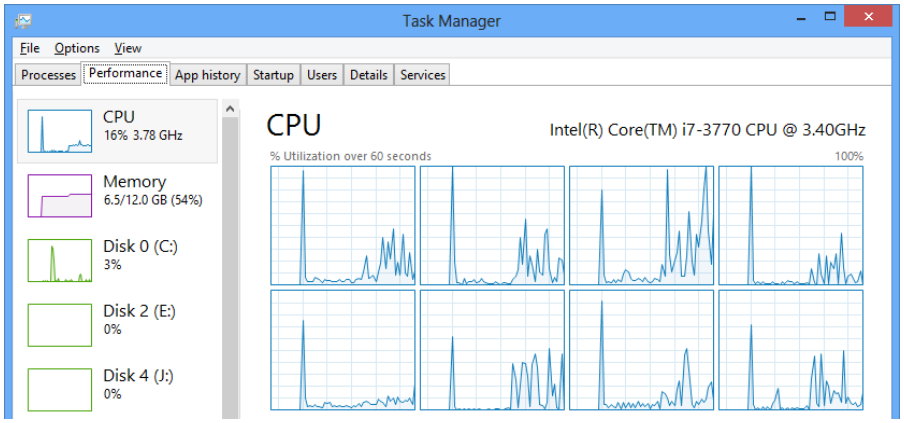
http://www.cplusplus.com/reference/thread/

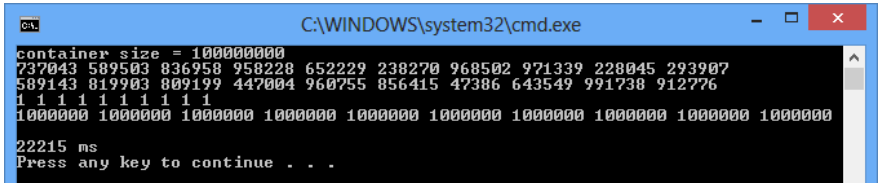
```cpp
void thread_func()
{
    std::stringstream ss;
    ss << "inside thread with id "<< std::this_thread::get_id() << "..." << std::endl;
    std::cout << ss.str();
}


std::thread t1(thread_func);
std::thread t2{ thread_functor() };

t1.join();

t2.join();
//if (t2.joinable()) t2.detach();
try
{
    t2.join();
}
catch (std::system_error& e)
{
    auto err = e.code();
    if (err.value() == (int)std::errc::invalid_argument)
        std::cout << "The thread object is not joinable" << std::endl;
    else
        std::cout << e.what() << std::endl;
}
```

```cpp
template <class RandomAccessIterator>
void merge_sort(RandomAccessIterator first, RandomAccessIterator end)
{
    if (end - first <= 1) return;

    auto mid = std::distance(first, end) / 2;
    merge_sort(first, first + mid);
    merge_sort(first + mid, end);
    merge(first, end, mid);
}

#include <sstream>

template <class RandomAccessIterator>
void parallel_merge_sort(RandomAccessIterator first, RandomAccessIterator end)
{
    if (end - first <= 1) return;

    auto size = std::distance(first, end);
    auto nothreads = std::thread::hardware_concurrency();

    std::vector<std::thread> threads;
    auto it = first;

    auto f = [](RandomAccessIterator f, RandomAccessIterator e)
    {
        std::stringstream ss;
        ss << std::distance(f, e) << " " << &f << " " << &e << std::endl;
        std::cout << ss.str();
        merge_sort(f, e);
    };

    auto stride = static_cast<unsigned>(size * (1.0 / nothreads));
    for (unsigned i = 1; i < nothreads; ++i, it += stride)
    {
        std::thread t(f, it, it + stride);
        threads.push_back(std::move(t));
    }
    std::thread t(f, it, end);
    threads.push_back(std::move(t));
    for (auto& th : threads)
        th.join();

    merge(first, end, std::distance(first, end) / 2);
}
```

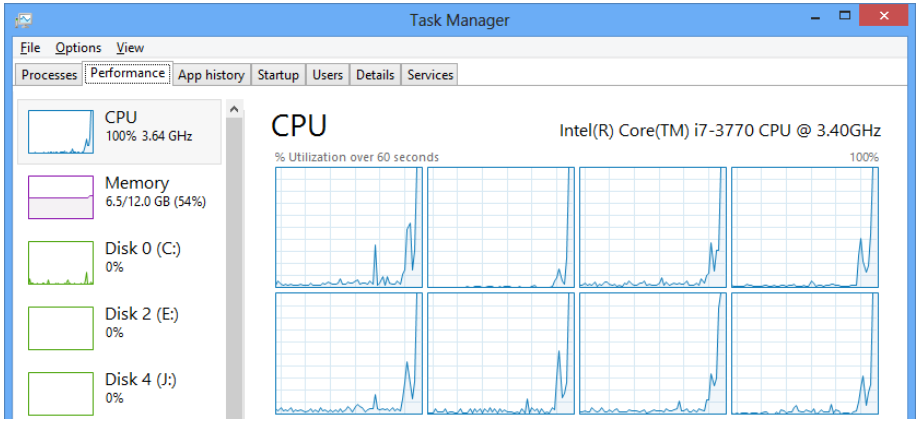
# *Embarrassingly Parallel*

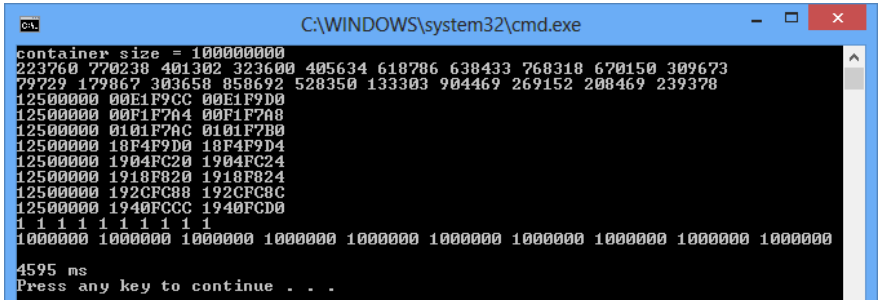
## Serial




## Paralelo



# #include <mutex>

header
## <mutex>

**Mutex**

Header with facilities that allow *mutual exclusion* (mutex) of concurrent execution of critical sections of code, allowing to explicitly avoid data races.

It contains *mutex types*, *lock types* and specific *functions*:

- **Mutex types** are *lockable types* used to protect access to a *critical section* of code: *locking* a mutex prevents other threads from locking it (exclusive access) until it is *unlocked*: `mutex`, `recursive_mutex`, `timed_mutex`, `recursive_timed_mutex`.
- **Locks** are objects that manage a mutex by associating its access to their own lifetime: `lock_guard`, `unique_lock`.
- **Functions** to lock mutiple mutexes simultaneously (`try_lock`, `lock`) and to directly prevent concurrent execution of a specific function (`call_once`).

### Classes

**Mutexes**

| | |
|---|---|
| **mutex** | Mutex class (class ) |
| **recursive_mutex** | Recursive mutex class (class ) |
| **timed_mutex** | Timed mutex class (class ) |
| **recursive_timed_mutex** | Recursive timed mutex (class ) |

**Locks**

| | |
|---|---|
| **lock_guard** | Lock guard (class template ) |
| **unique_lock** | Unique lock (class template ) |

**Other types**

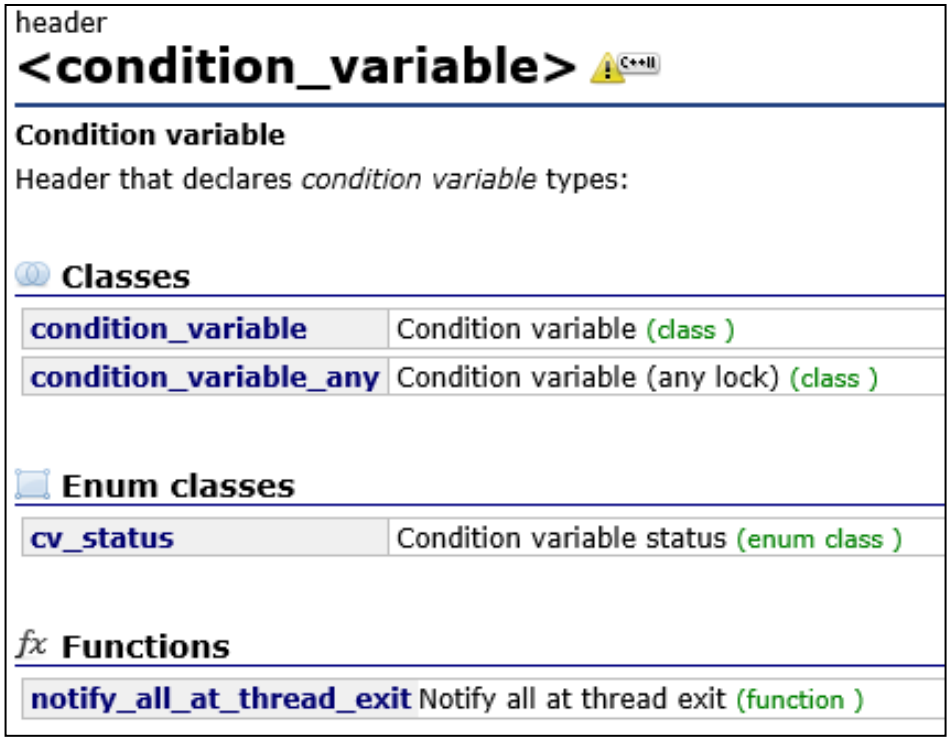
| | |
|---|---|
| **once_flag** | Flag argument type for call_once (class ) |
| **adopt_lock_t** | Type of adopt_lock (class ) |
| **defer_lock_t** | Type of defer_lock (class ) |
| **try_to_lock_t** | Type of try_to_lock (class ) |

### Functions

| | |
|---|---|
| **try_lock** | Try to lock multiple mutexes (function template ) |
| **lock** | Lock multiple mutexes (function template ) |
| **call_once** | Call function once (public member function ) |

http://www.cplusplus.com/reference/mutex/

# #include <condition_variable>



header

## <condition_variable>

**Condition variable**

Header that declares *condition variable* types:

### Classes

| | |
|---|---|
| **condition_variable** | Condition variable (class ) |
| **condition_variable_any** | Condition variable (any lock) (class ) |

### Enum classes

| | |
|---|---|
| **cv_status** | Condition variable status (enum class ) |

### Functions

| | |
|---|---|
| **notify_all_at_thread_exit** | Notify all at thread exit (function ) |

http://www.cplusplus.com/reference/condition_variable/

```cpp
#include <mutex>
#include <condition_variable>
#include <queue>

template<typename T>
struct blocking_queue final
{
    void enqueue(const T& item)
    {
        std::lock_guard<std::mutex> lock(CR_);
        Q_.push(item);
        NonEmpty_.notify_one();
    }

    T dequeue()
    {
        std::unique_lock<std::mutex> lock(CR_);
        NonEmpty_.wait(lock, [this](){ return !Q_.empty(); });
        //or
        //while (Q_.empty()) NonEmpty_.wait(lock);
        T temp = Q_.front();
        Q_.pop();
        return temp;
    }

private:
    std::condition_variable NonEmpty_;
    std::mutex CR_;
    std::queue<T> Q_;
};
```

```cpp
blocking_queue<int> q;

//consumer
auto deq_f = std::async([&q](){
    int value;
    while ((value = q.dequeue()) != 0)
        std::cout << value << " ";
});

//producer
std::default_random_engine engine(static_cast<unsigned>(std::time(nullptr)));
std::uniform_int_distribution<int> rnd(1, 100000);

for (int i = 0; i < 10; ++i)
{
    for (int j = 0; j < 10; ++j) q.enqueue(rnd(engine));
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

std::this_thread::sleep_for(std::chrono::seconds(3));
q.enqueue(0);
```

# #include <future>

header
## <future>

**Future**

Header with facilities that allow asynchronous access to values set by specific providers, possibly in a different thread.

Each of these providers (which are either `promise` or `packaged_task` objects, or calls to `async`) share access to a *shared state* with a `future` object: the point where the *provider* makes the *shared state* ready is synchronized with the point the `future` object accesses the *shared state*.

### Classes

**Providers**

| | |
|---|---|
| **promise** | Promise (class template ) |
| **packaged_task** | Packaged task (class template ) |

**Futures**

| | |
|---|---|
| **future** | Future (class template ) |
| **shared_future** | Shared future (class template ) |

**Other types**

| | |
|---|---|
| **future_error** | Future error exception (class ) |
| **future_errc** | Error conditions for future objects (enum class ) |
| **future_status** | Return value for timed future operations (enum class ) |
| **launch** | Launching policy for async (enum class ) |

### Functions

**Providers**

| | |
|---|---|
| **async** | Call function asynchronously (function template ) |

**Other functions**

| | |
|---|---|
| **future_category** | Return future category (function ) |

http://www.cplusplus.com/reference/future/

```cpp
//future and async
std::future<int> f =
    std::async([]() -> int {
    std::this_thread::sleep_for(std::chrono::seconds(2));
    return 100;
});

std::cout << f.get() << std::endl;
```

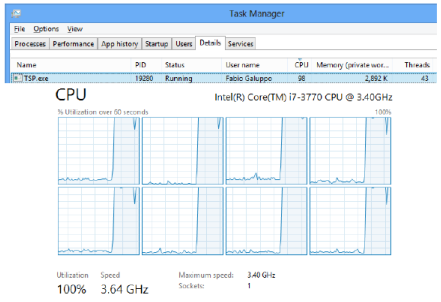
```cpp
auto g = [](std::promise<int>& p, int value) {
    if (value <= 0)
    {
        auto e = std::make_exception_ptr(std::runtime_error("value less
        p.set_exception(e);
        return;
    }

    std::this_thread::sleep_for(std::chrono::seconds(2));
    p.set_value(value * 100);
};

//promise
std::promise<int> p1, p2;
std::thread t1(g, std::ref(p1), 100);
std::thread t2(g, std::ref(p2), -1);

auto f1 = p1.get_future();
auto f2 = p2.get_future();

f1.wait();
f2.wait();

std::cout << f1.get() << std::endl;
try
{
    std::cout << f2.get() << std::endl;
}
catch (const std::exception& e)
{
    std::cout << "exception caught: " << e.what() << std::endl;
}

t1.join();
t2.join();
```

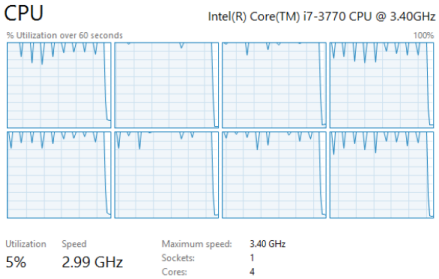
```cpp
//packaged_task
auto h = [](int a, int b)
{
    std::this_thread::sleep_for(std::chrono::seconds(2));
    int result = a * b;
    if (result <= 0)
        throw std::runtime_error("result less than or equal to 0");
    return result;
};
std::packaged_task<int(int, int)> pt1(h);
std::future<int> f3 = pt1.get_future();
std::thread t3(std::move(pt1), 10, 90);

std::packaged_task<int(int, int)> pt2(h);
std::future<int> f4 = pt2.get_future();
std::thread t4(std::move(pt2), 10, -90);
t3.detach();
t4.detach();

std::cout << f3.get() << std::endl;
try
{
    std::cout << f4.get() << std::endl;
}
catch (const std::exception& e)
{
    std::cout << "exception caught: " << e.what() << std::endl;
}
```

# *Wrapping up*

```
//TSP -> NN -> Generations( g, ForkJoin ( n, SA -> 2-OPT ) ) -> TSP'
void Pipeline1(tsp_class& tsp_instance, unsigned int number_of_tasks, unsigned int number_of_generations)
{
	#pragma region "PipelineConfiguration"
	auto a = Args<General_args_type>(make_General_args(number_of_generations, number_of_tasks));
	auto sa = Args<SA_args_type>(make_SA_args(1000.0, 0.00001, 0.999, 400));
	auto aco = Args<ACO_args_type>();
	auto ga = Args<GA_args_type>();

	const char* pipeline_description = "TSP -> NN -> Generations( g, ForkJoin ( n, SA -> 2-OPT ) ) -> TSP'";
	display_args(pipeline_description, a, sa, aco, ga);

	auto g = a[0].number_of_iterations_or_generations;
	auto n = a[0].number_of_tasks_in_parallel;
	auto _TSP = TSP(just(tsp_instance));
	auto _DisplayInput = Display("TSP INPUT", DisplayFlags::All);
	auto _NN = Measure(NN(), Display("NEAREST NEIGHBOUR", DisplayFlags::EmitMathematicaGraphPlot));
	auto _SA_2OPT = Chain(SA(sa[0].initial_temperature, sa[0].stopping_criteria_temperature,
	                         sa[0].decreasing_factor, sa[0].monte_carlo_steps), _2OPT());
	auto _ForkJoin = [](unsigned int n, TSP::transformer_type map_fun){ return Measure(ForkJoin(n, map_fun)); };
	auto _DisplayOutput = Display("TSP OUTPUT", DisplayFlags::EmitMathematicaGraphPlot);
	#pragma endregion

	//TSP -> NN -> Generations( g, ForkJoin ( n, SA -> 2-OPT ) ) -> TSP'
	auto result = _TSP
	              .map(_DisplayInput)
	              .map(_NN)
	              .map(Generations(g, _ForkJoin(n, _SA_2OPT)))
	              .map(_DisplayOutput);
}
```

http://bit.ly/1hoODIh

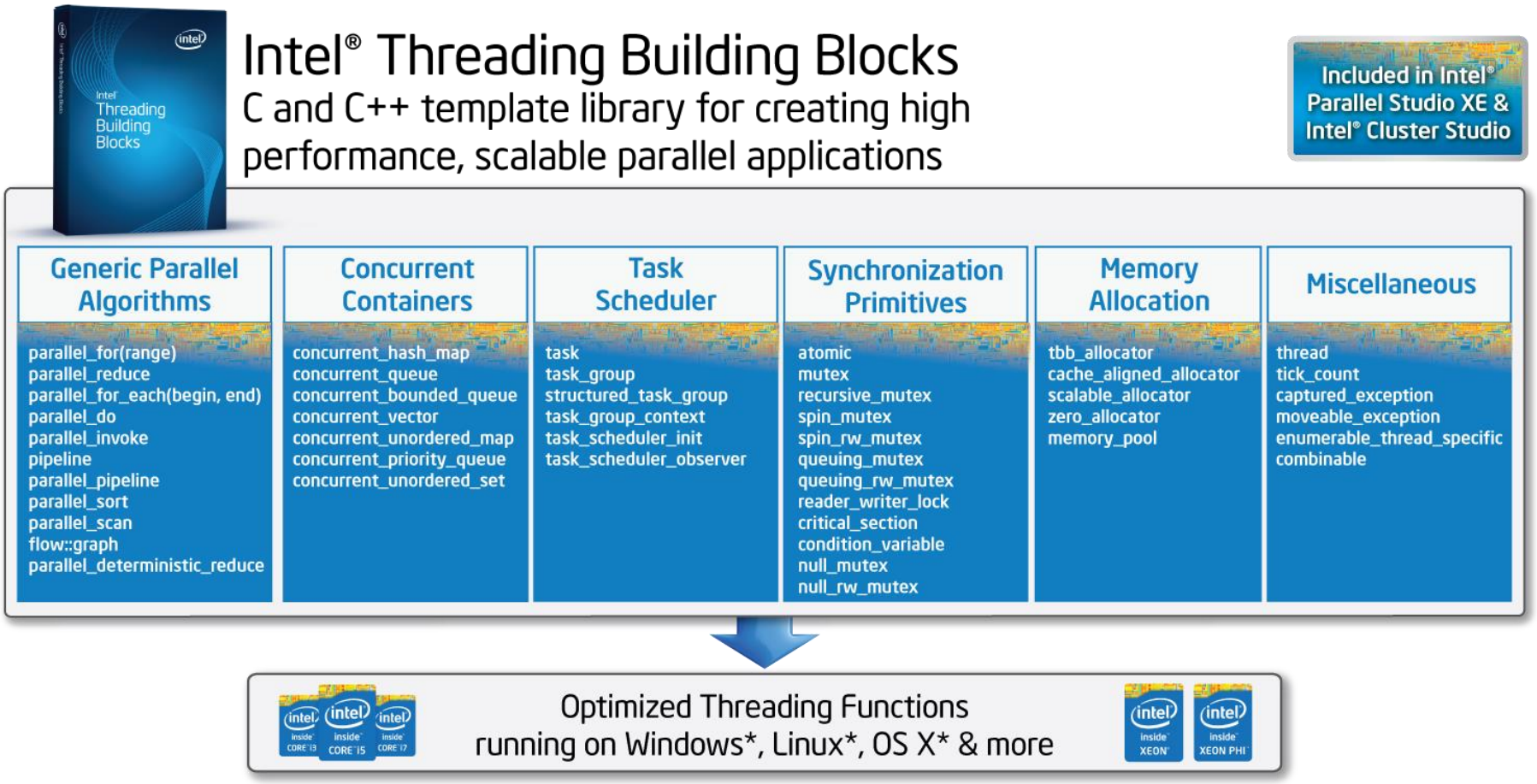



http://bit.ly/1ppSmKN

```
struct ForkJoin
{
	typedef TSP::transformer_type F;

	ForkJoin(unsigned int number_of_tasks_in_parallel, F f, bool propagate_inferior_results = false) :
		NumberOfTasksInParallel_(number_of_tasks_in_parallel),
		Func_(f),
		PropagateInferiorResults_(propagate_inferior_results)
	{
	}

	TSP operator()(TSP::T t)
	{
		if (0 == NumberOfTasksInParallel_)
			return TSP(t);

		std::vector<std::future<tsp_async_state>> tsp_solutions;

		std::cout << "START SOLUTION:" << std::endl;

		auto tsp_instance = ref(t);
		display_solution(tsp_instance);

		for (unsigned int i = 0; i < NumberOfTasksInParallel_; ++i)
		{
			//async
			tsp_solutions.push_back(std::async(std::launch::async, [&, i, tsp_instance]
			{
				Maybe a = just(tsp_instance);
				std::unique_ptr<ForkJoin_args> args(new ForkJoin_args(i, NumberOfTasksInParallel_));
				a.second = args.get();
				return tsp_async_state(Func_(a));
			}));
		}

		std::cout << "CANDIDATE SOLUTIONS:" << std::endl;
		std::vector<tsp_class> candidate_solutions;
		int i = 0;
		for (auto& tsp_solution : tsp_solutions)
		{
			//async result
			auto candidate = tsp_solution.get();
			if (has(candidate.state))
			{
				candidate_solutions.push_back(ref(candidate.state));
				display(ref(candidate.state), false);
			}
		}
	}
```

# Intel Threading Building Blocks (TBB)




https://software.intel.com/en-us/intel-tbb

https://www.threadingbuildingblocks.org/

# Visual C++ ConcRT e Parallel Patterns Library (PPL)

## PPL Compatibility

Intel® Threading Building Blocks (Intel® TBB) 2.2 introduces features based on joint discussions between the Microsoft Corporation and Intel Corporation. The features establish some degree of compatibility between Intel® TBB and Microsoft Parallel Patterns Library (PPL) development software.

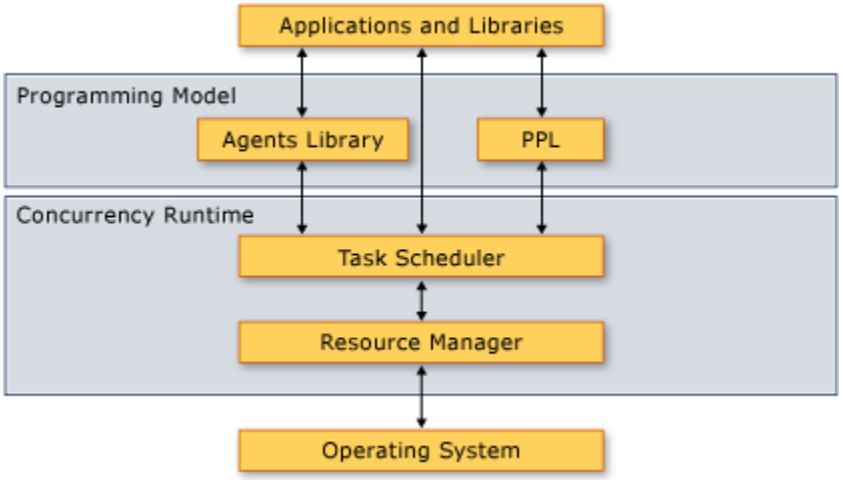
Each feature appears in namespace `tbb`. Each feature can be injected into namespace Concurrency by including the file "`tbb/compat/ppl.h`". Following is the list of features:

The following table lists the features and provides links to additional information.

| Feature | Link |
|---|---|
| parallel_for(*first*,*last*,*f*) | parallel_for Template Function |
| parallel_for(*first*,*last*,*step*,*f*) | parallel_for Template Function |
| `parallel_for_each` | parallel_for_each Template Function |
| `parallel_invoke` | parallel_invoke Template Function |
| `critical_section` | critical_section |
| `reader_writer_lock` | reader_writer_lock Class |
| `task_handle` | task_handle Template Class |
| `task_group_status` | task_group_status Enum |
| `task_group` | task_group Class |
| `make_task` | make_task Template Function |
| `structured_task_group` | structured_task_group Class |
| `is_current_task_group_cancelling` | is_current_task_group_canceling Function |
| `improper_lock` | Specific Exceptions |
| `invalid_multiple_scheduling` | Specific Exceptions |
| `missing_wait` | Specific Exceptions |

For `parallel_for`, only the variants listed in the table are injected into namespace Concurrency.

**CAUTION**
Because of different environments and evolving specifications, the behavior of the features can differ between the Intel® TBB and PPL implementations.




http://msdn.microsoft.com/en-us/library/ee207192.aspx

The PPL provides the following features:

- *Task Parallelism*: a mechanism to execute several work items (tasks) in parallel
- *Parallel algorithms*: generic algorithms that act on collections of data in parallel
- *Parallel containers and objects*: generic container types that provide safe concurrent access to their elements

http://msdn.microsoft.com/en-us/library/dd492418.aspx

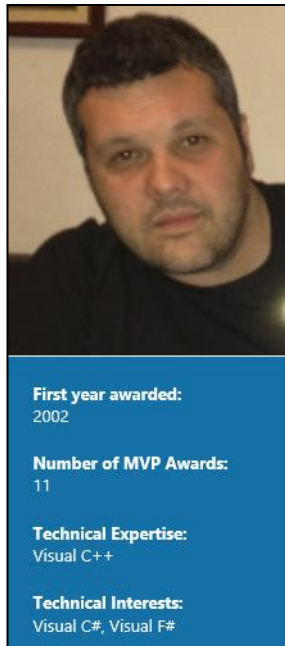
# Introdução a *Multi-threading* com C++ 11

(C++ 11 == ISO/IEC 14882:2011)


Fabio Galuppo, M.Sc.

http://fabiogaluppo.com

fabiogaluppo@acm.org

**First year awarded:**
2002

**Number of MVP Awards:**
11

**Technical Expertise:**
Visual C++

**Technical Interests:**
Visual C#, Visual F#