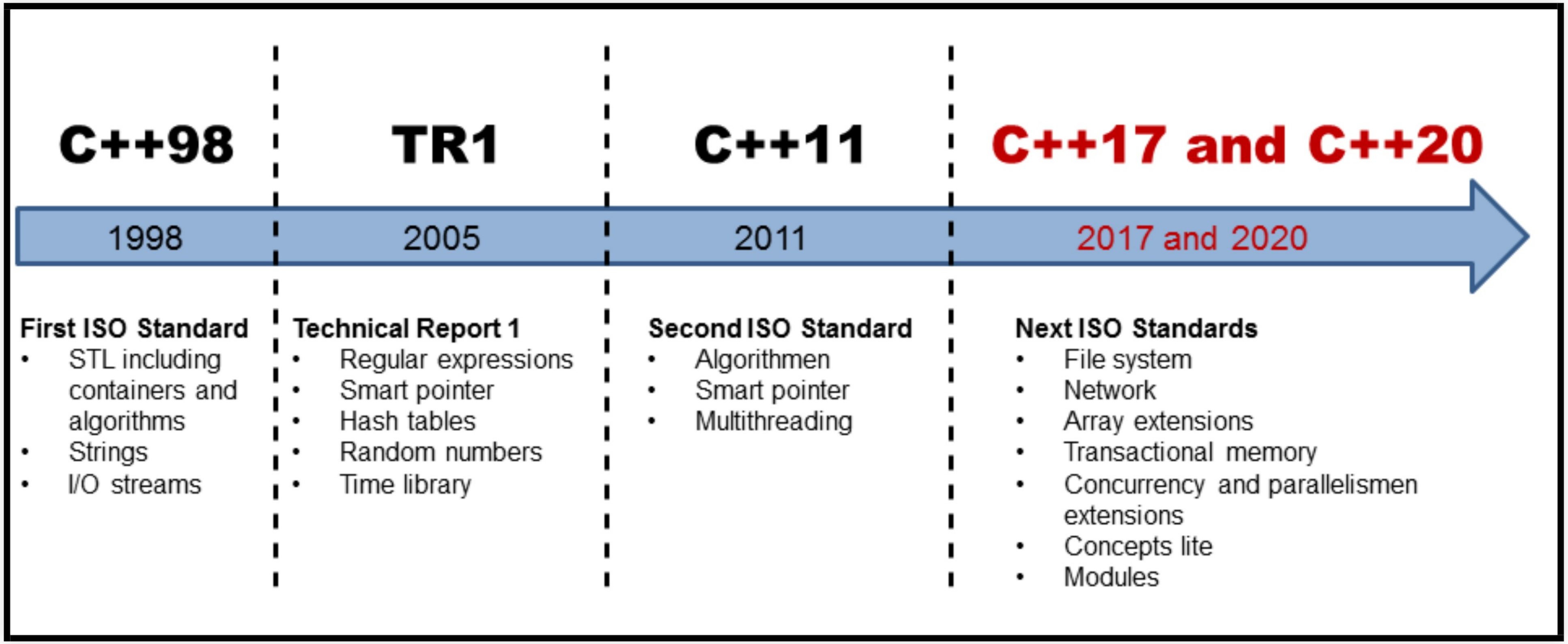
C++-{1, 2}z

C++-{11, 14, 17, 20}

Jonathan Muller / @bhaal22

History


C++98
TR1
C++11
C++17 and C++20
1998
2005
2011
2017 and 2020
First ISO Standard
• STL including containers and algorithms
• Strings
• I/O streams
Technical Report 1
• Regular expressions
• Smart pointer
• Hash tables
• Random numbers
• Time library
Second ISO Standard
• Algorithmen
• Smart pointer
• Multithreading
Next ISO Standards
• File system
• Network
• Array extensions
• Transactional memory
• Concurrency and parallelismen extensions
• Concepts lite
• Modules

Motivations

- No updates of the language / standard library since 2003
- Increase type safety by providing safer alternatives to earlier unsafe techniques
- Increase performance and the ability to work directly with hardware
- Implement zero-overhead principle
- Make C++ easy to teach and to learn without removing any utility needed by expert programmers

Motivations: personal thoughts

- make c++ fun again
- try to get back developpers to use this language
- other languages have a shorter lifecycle: javascript, java, c#
- c# is born in 2002 and currently c# 7 is implemented
- try to improve speed of writing c++ code
- since 2011, language / standard library improves every 3 years

Core language updates

- compile time improvement: extern template
- Initializer lists
- type inference: **auto** keyword and **decltype**
- lambda functions: syntaxic sugar around functors

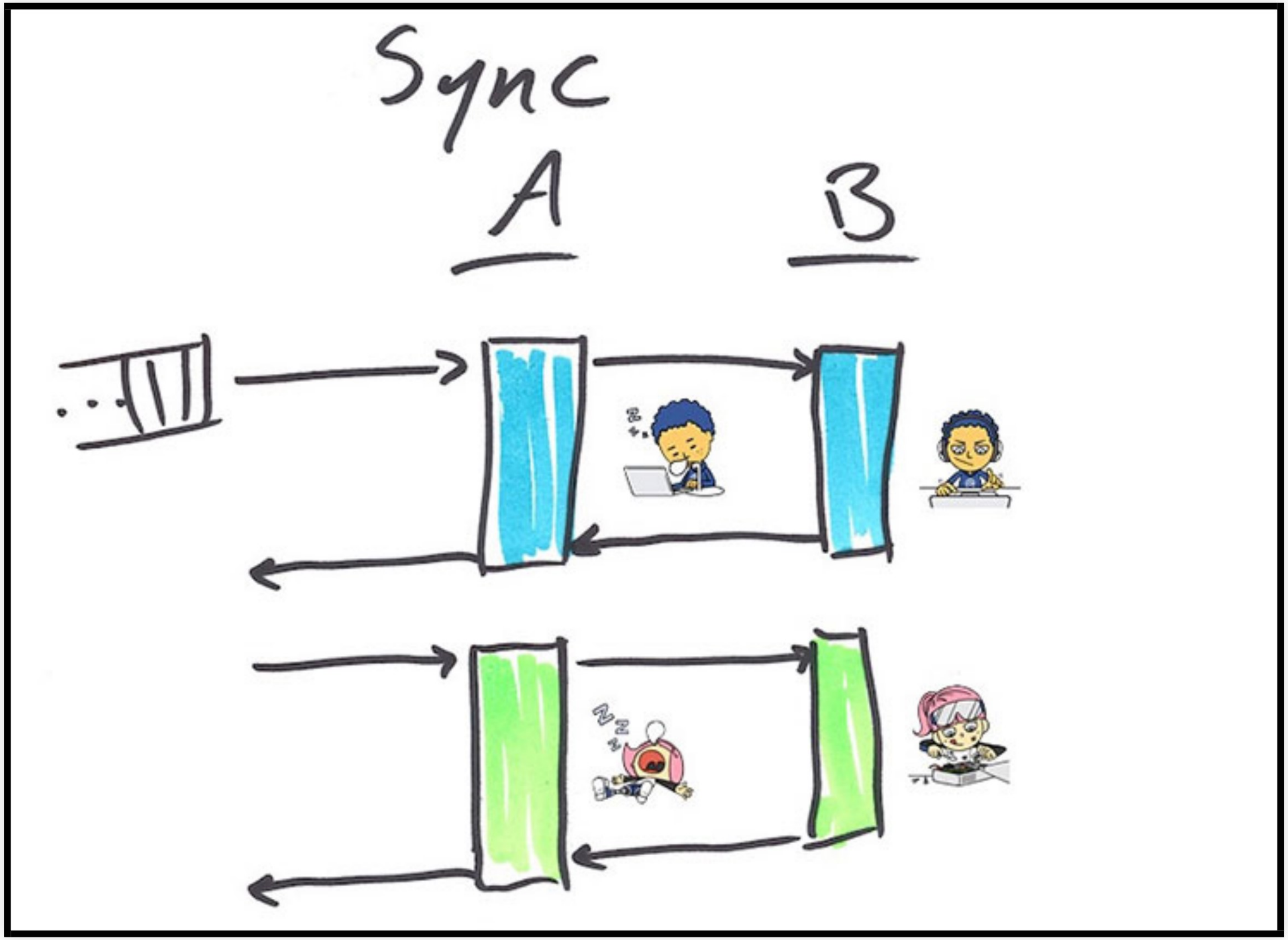
Core language updates: Class hierarchy

- **override** keyword: tell compile to look in inheritance tree for a virtual method with the same signature
- **final** keyword: class definition / method definition

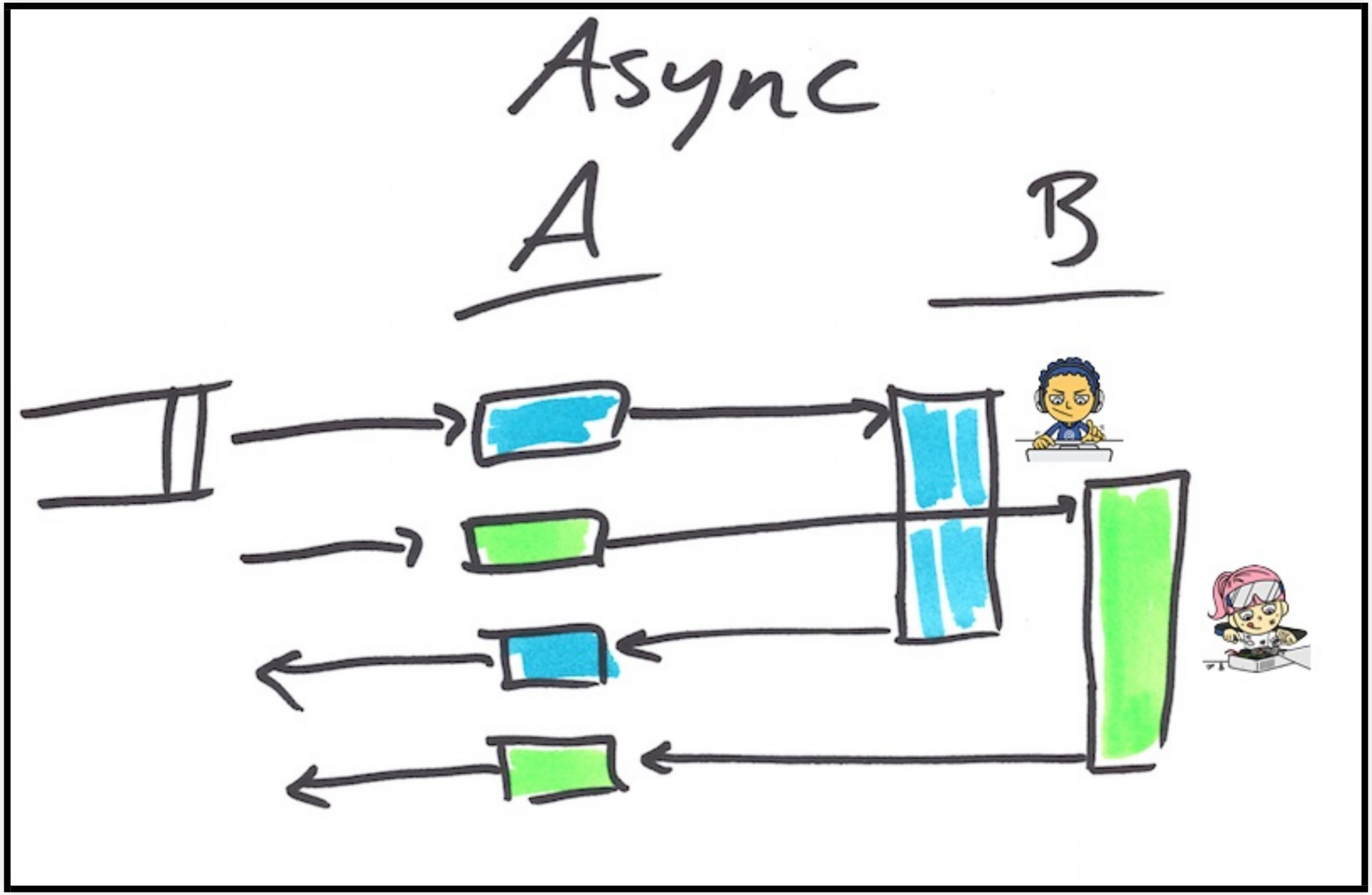
Core language updates: misc

- strongly typed **nullptr**
- stronglty typed enums
- string literals: u8 (utf-8), u (utf-16), U (utf-32)
- smart pointers: std::unique_ptr, std::shared_ptr, std::weak_ptr

standard library: threading



standard library: threading



standard library: threading

- abstraction initially defined in boost
- std::thread
- std::mutex
- std::condition_variable

standard library: asynchronous

- std::thread : low level asynchronous mechanism
- a task is something a developper wants to be done (e.g. a computation)
- a task uses a thread as support for asynchronous operations
- std::async

Javascript future / promise: chain defers

```
// Defered chaining
return httpGet(currentModule.fullUrl).then(function (res){
    return loadBody(res);
}, currentModule.httpGetError).then(function (webData) {
    return envPromise(webData);
}).then(function (window){
    return extractData(window);
}, currentModule.envError);
```

C# aync / await

```
public async Task<user> GetUser(int userId)
{
    // Try Do something smart
}

public static async Task<user[]> GetUsers(IEnumerable<int> userId
{
    var getUserTasks = userIds.Select(id => GetUser(id));
    return await Task.WhenAll(getUserTasks);
}
                        </int></user[]></user>
```

Two Tasks. One process. (I'll give you something in the Future. I Promise.)

- promise / futur paire = machanism to get data between threads
- assume you have no need for coordination between threads and no shared data
- compute a single function in parallel for two different initial values
- futures = result of an asynchronous result provider
- std::promise, std::packaged_task, and std::async

Differences of the different interfaces: **std::promise**

- std::promise. The most flexible way to provide a value for a future. Computation is performed independently from the promise object and the result is simply passed through the object to the future using the set_value() method.

Differences of the different interfaces: **std::packaged_task**

- std::packaged_task. The second most flexible way to provide a value for a future. The constructor takes a function and uses the return value of that function to set the value for the future. Since packaged_tasks must be explicitly invoked, they can be created and then assigned to be run on particular threads.

Differences of the different interfaces: **std::async**

- std::async. Provides the least flexibility, but also the simplest way to execute an asynchronous computation. The method takes a function and uses the return value of that function to set the value for the future.
- the primary distinction between std::async and std::packaged_task is that std::async automatically begins execution upon calling it.
- the caller has no control over where the task is scheduled to run (including on the current thread).

Limitations

- Cannot chain futures (std::future::then())
- Must wait for C++20
- Available in TS experimental package

```
future<int> f1 = async([]() {return 123;});
future<string> f2 = f1.then([](future<int> f){ return f.get().to_

future<int> futures[] = {async([]() { return intResult(125); }),
                         async([]() { return intResult(456); })};
future<vector<future<int>>> any_f = when_any(begin(futures), end(

future<int> futures[] = {async([]() { return intResult(125); }),
                         async([]() { return intResult(456); })};
future<vector<future<int>>> all_f = when_all(begin(futures), end(
                        </vector<future<int></int></vector<future
```

- C++ Specification Repository