C++ Evolves!

Axel Naumann
PH-SFT, CERN, CH-1211 Geneva 23, Switzerland
E-mail: `axel@cern.ch`

Abstract. C++ is used throughout High Energy Physics. CERN participates in the development of its standard. There has been a major shift in standardization procedures that will be visible starting 2014 with an increase rate of new standardized features. Already the current C++11 has major improvements, also for coding novices, related to simplicity, expressiveness, performance and robustness. Other major improvements are in the area of concurrency, where C++ is now on par with most other high level languages. To benefit from these language improvements and from the massive improvements in compiler technology for instance in usability, access to current compilers is crucial. Use of current C++ compiled with current compilers can considerably improve C++ for the HEP physicist community.

1. Introduction
High Energy Physics relies on C++ for collaborative development of high performance code. But C++ is not the language it used to be. In fact, the process of upgrading C++ itself has changed: a massive influx of features is foreseen over the next couple of years, making the language itself and especially its standard library much more dynamic. At the same time, compiler and library vendors have adapted. They publish new versions regularly, often implementing new C++ features even proactively. High Energy Physics will have to adapt if it wants to benefit from the improvements, not only in C++ but also in the compilers themselves. Given that the improvements are often explicitly targeted to simplicity and robustness, we cannot afford to simply ignore this progress.

2. The C++ Standard
The definition of C++ is governed by the International Organization for Standardization through the C++ Standards Committee JTC1/SC22/WG21 [1]. To date there have been two major revisions of the standard, one from 1998 ("C++98") and one from 2011 ("C++11"). Most current C++ code in High Energy Physics uses features defined in the 1998 standard. The current draft of the standard is maintained in github and is freely available [2]; the normative standard text itself is copyrighted and not freely available.

2.1. Structure of the Committee
To accelerate the delivery of new features, the standardization committee has created several topical Study Groups as shown in fig. 1. Each group investigates and coordinates the development of new features for a specific domain. The most important Study Groups for HEP are Concurrency (that includes vectorization) and Transactional Memory, as well as standard library extensions from Filesystem, Networking, and Numerics. There are also study groups

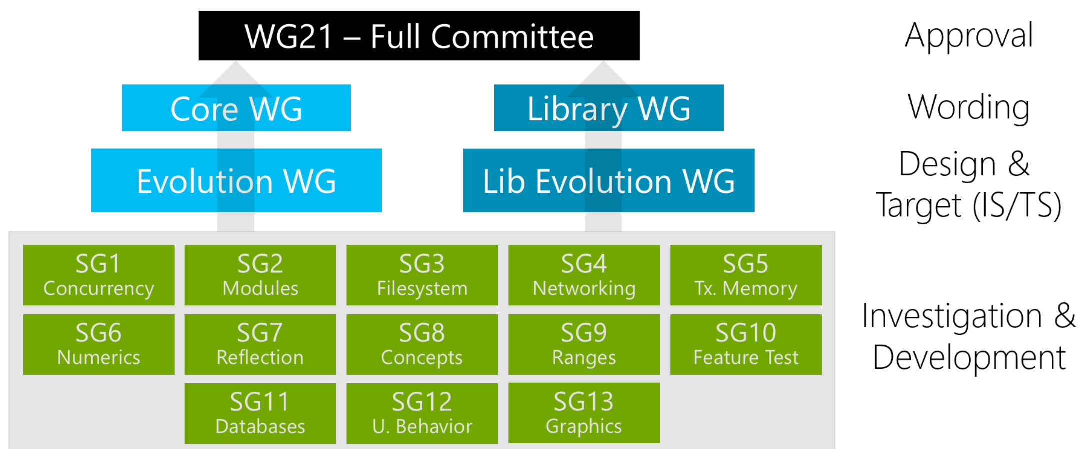

Figure 1: Committee structure including current study groups (from `isocpp.org`).

that will have a fundamental impact on the way we use C++. Modules, for instance, could replace bundles of header files, both conceptually and for the compilation process; Reflection could add type information currently provided usually through ROOT's dictionary system.

Proposals to the committee are discussed during the approximately three committee meetings per year. There are typically 10 – 30 participants in each group that meet to discuss and cast an informal vote (during "straw polls") on the fate of these proposals. Once the proposals have been reviewed by the working groups they are voted into a standardization text by the full committee. Being an ISO committee, each national standardization body has one vote. CERN is part of the Swiss delegation that currently consists of three members.

2.2. Standard Publication Sequence

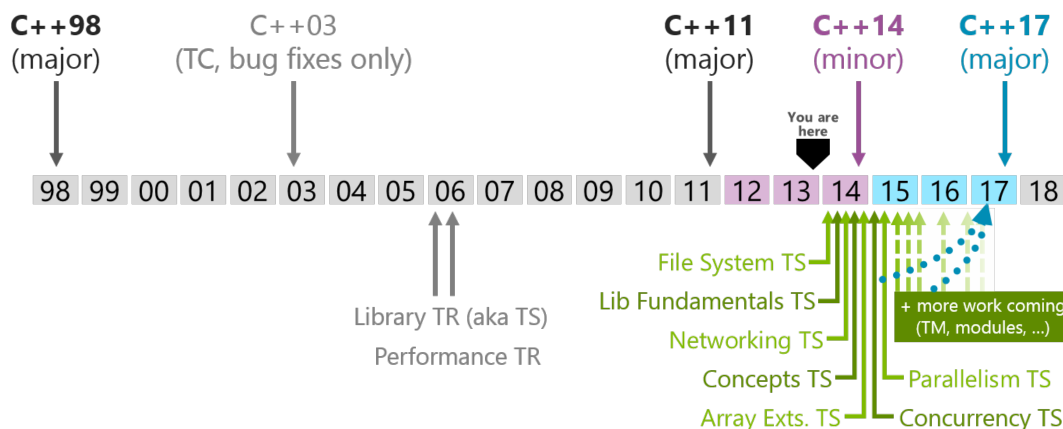

Figure 2: Past and expected future timeline of standard documents (from `isocpp.org`).

Figure 2 shows the past and expected future standard documents. Some of them are created by study groups, then reviewed by the committee's "regular" working groups, and then published as separate standardization documents ("Technical Specifications", TS) and later incorporated in the standard. This independent publication process allows compiler and library vendors to

implement the features early; many have pledged to offer them as soon as they are published. At the time of writing, the Filesystem Technical Specification is already under review by the national bodies and expected to be published in the first half of 2014.

Compared to the standardization progress before 2011, there is now a dramatic increase in the rate of change to C++ and its standard library. These changes will rapidly be implemented in forthcoming versions of compilers. Already, GCC and clang implement a fair fraction of the next standard revision that will be published as "C++14' in 2014.

2.3. High Energy Physics Participation

Until a few years ago, Fermilab was actively involved in the standardization process, leaving many traces in C++11. Once its active participation ceased, CERN became a member, also on explicit request by Bjarne Stroustrup, the inventor of C++. CERN's participation is beneficial for both High Energy Physics and C++ itself for the following reasons:

Given HEP's investment in C++, in terms of a very large code base and widespread community of developers educated in C++, we have a natural interest in the future of the language:

- Our programs must work also with future versions of C++.
- The need to reach optimal performance performance and ease of use should be represented in the committee.
- Our software is to a large extent not written by software specialists.
- Thousands of people collaborate through header files with their experiments' frameworks and each others' analysis code.
- We perform performance / throughput critical operations for instance in reconstruction and simulation.
- We use C++ in a heterogeneous language environment.
- We store petabytes of data defined through C++ [3, 4]
- We use C++ as an interpreted language [5].

Additionally, users are a scarce resource in the committee: most of its members are software or language experts, compiler or library vendors, or C++ wizards. Novice and "general" users are underrepresented. Given HEP's wide range of usage of C++ (for instance for hardware control, number crunching and graphical user interfaces) and our similarly wide range of fields of expertise we should be able to contribute to many areas of the standardization process.

3. Language Features of C++11

While the current C++ standard facilitates complex meta programming and other software expert needs (e.g. lambdas, `constexpr`, user defined literals), its key improvements are in the area of code simplicity and robustness.

3.1. Simplicity

One of the most cumbersome features of the C++98 standard were iterations over collections, as illustrated in this code fragment:

```
const std::map<std::string, std::vector<MyClass> >& m = ...;
for (std::map<std::string, std::vector<MyClass> >::const_iterator
       i = m.begin(), e = m.end(); i != e; ++i) {
```

Writing this for-loop correctly is very verbose and not trivial. Writing a performance-optimal version of this for-loop is even less trivial in that it would cache the return value of `end()` and use the prefix increment to avoid possible temporaries. Current C++ argues that the type of the variables can be defined by the type of their initializers, shortening the loop to:

```
for (auto i = begin(m), e = end(m); i != e; ++i) {
```

The code example also employs the generic `begin()`, `end()` functions provided by current C++. But the need for performance optimization did not change in this version of the code. Adding the fact that this is a very common construct motivates an even simpler version:

```
for (auto i: m) {
```

Just as for the previous examples, this loop iterates over all elements of the collection `m` with an iterator variable `i`. It nicely demonstrates one of the major improvements in C++, i.e. making code much more readable. Other examples are constructor delegation and deletion.

3.2. Robustness

Another essential improvement in C++11 are the features that enable robust code, for instance by clarifying ownership of pointers. In this scheme, raw "C-style" pointers signal non-ownership, i.e. `delete` should rarely be needed. Instead, owning pointers are of type `std::unique_ptr`. A `std::unique_ptr` can hand over (move) ownership to another `std::unique_ptr`. The common request for garbage collection in C++ can often be satisfied by using the reference counted `std::shared_pointer`.

Use of these types in interfaces can dramatically reduce the number of memory errors, by enforcing ownership rules on the type layer. In particular, the `std::unique_ptr` can be implemented in a nearly or completely performance-neutral way.

3.3. Performance

Containers that retrieve elements based on a hash of the key are finally available in the standard library. For instance, almost all uses of `std::map<std::string,X>` should likely be migrated to the generally more performant `std::unordered_map<std::string,X>`. The same holds for `std::set<std::string>` versus `std::unordered_set<std::string>`.

Container initialization with a set of elements known at compile time was excessively verbose to write in C++98 and also inefficient, as the initialization incurred a runtime cost:

```
std::vector<int> v;
v.push_back(12);
v.push_back(42);
v.push_back(17);
v.push_back(12);
v.push_back(9);
```

This often drove developers to use C-style arrays even though they do not offer size checks nor storage management and thus incur a higher risk of memory errors. Initializer lists make initialization, for instance of standard containers, both a compile-time operation and much more compact:

```
std::vector<int> v{12,42,17,12,9};
```

Other examples of performance improvements in C++ are `constexpr` (functions evaluated at compile time) and move semantics that prevent copying of data.

4. Migration to C++11

While the current 2011 standard is mostly backward compatible, in that old code can be compiled while applying the current rules, code built under the 2011 standard revision cannot generally be linked against code compiled with the previous revision. To make this explicit, several compilers, as for instance GCC, clang, the Intel Compiler, require an additional flag to turn on the new rules, typically `-std=c++11`. ROOT's entire source code of about 3 million lines required only two types of change: initializer lists now have stricter type checks and preprocessor macro

pasting shows a different behavior when string literals are involved. For instance clang's error messages will helpfully spell out the required changes, making the support of the new standard a straight-forward operation.

4.1. Current Adoption of C++11 in High Energy Physics
Our recent survey of frameworks' and experiments' plans of employing C++11 showed a wide range of adoption. While NOvA physicists already use C++11 in their analysis code, most of the LHC experiments (CMS, ATLAS, LHCb) and Belle II are currently using it just in their frameworks. For most of the LHC experiments, the use of C++11 is physics analysis is expected to start in earnest when data taking resumes after the Long Shutdown 1 (LS1). Others, such as FairROOT and ALICE, have validated their code to compile with C++11, whereas RHIC uses the subset of C++ known as "C++0x" available in GCC versions up to 4.6 with plans to move to C++11 in 2015. Looking at bug reports and a survey of physicists' use of C++11, we conclude that the current usage in analysis code seems to be minimal.

5. Deployment of Current C++
One of the main reasons for NOvA's use of C++11 in analysis code is the availability of a modern compiler: art [6], the framework used by NOvA, is distributed together with a current version of GCC. Other deployment mechanisms involve cvmfs [7]. Old compilers, such as those available by default on Scientific Linux 5 (GCC 4.1) and 6 (GCC 4.4), have no or very limited support for C++11.

5.1. Compiler Diagnostics
In a response to requests for what C++ should improve, many physicists asked for improved diagnostics, even though they are not part of the C++ standard. This is a known issue for the "old generation" of compilers. As a reaction to the advent of clang with its much more expressive diagnostics, also GCC's diagnostics have been dramatically improved in recent versions.

As a typical example for the intricacies of diagnosing incorrect C++ code, the following code wrongly passes both an iterator and a const iterator to `std::find`:

```
#include <vector>
#include <algorithm>

typedef std::vector<double> MyV_t;
MyV_t Vec;
const MyV_t& ConstVec = Vec;

void f() {
   std::find(Vec.begin(), ConstVec.end(), 12);
}
```

Even the rather recent GCC 4.6 would diagnose the problem as shown here:

```
T.C: In function 'void f()':
T.C:9: error: no matching function for call to 'find(__gnu_cxx::
    __normal_iterator<double*, std::vector<double, std::allocator<double> > >,
    __gnu_cxx::__normal_iterator<const double*, std::vector<double, std::
    allocator<double> > >, int)'
```

A current clang (version 3.4 trunk 192279) will issue the report shown in Fig. 3 for the same code. Here, the color highlighting is part of the diagnostics issued by clang. Correcting the error now becomes very simple.

```
T.C:9:4: error: no matching function for call to 'find'
   std::find(Vec.begin(), ConstVec.end(), 12);
   ^~~~~~~~~
/usr/include/c++/4.6/bits/stl_algo.h:4394:5: note: candidate template
      ignored: deduced conflicting types for parameter '_InputIterator'
      ('__normal_iterator<double *, [...]>' vs.
      '__normal_iterator<const double *, [...]>')
    find(_InputIterator __first, _InputIterator __last,
    ^
1 error generated.
```

Figure 3: Diagnostics reported by clang 3.4

6. Concurrency Features of C++

Due to the ubiquity of concurrent concepts in modern code, C++ language and library development focus on offering concurrency features. Several of them are available today and several interesting concepts are being investigated for future revisions.

6.1. Operation System Abstractions

Current C++ offers abstractions of all standard threading and synchronization tools offered by modern operating systems: `std::thread`, `std::mutex`, `std::lock_guard` and `std::condition_variable` are the most prominent examples, with obvious counterparts in the operation system implementations. They can replace non-standard library abstractions, for instance in boost and ROOT. ROOT, for instance, plans to migrate the implementations of its abstractions to those provided by C++ whenever possible.

6.2. Tasks

Tasks are a high-level concurrent concept compared to operating system abstractions; they represent a function to be run independently from the main thread. Scheduling and mapping of tasks to the underlying operation system building blocks are left to the runtime system; the developer does not need to implement them. C++ provides `std::async()` to launch a task.

6.3. Futures

Futures are a handle to a value that does not need to be evaluated yet. As such it allows to delay the (possibly blocking) evaluation of the expression until it is really required, splitting the concepts of future-independent control flow from future-dependent parts of the code. Code using futures can hide the complexity of synchronization and give additional flexibility to the runtime. C++ implements them through `std::future`.

6.4. Memory

Current C++ offers `thread_local` variables that have separate storage for each thread, and `atomic_int` and alike for atomics protecting certain operations, for instance increments, from race conditions. Transactional Memory is a concept currently under study for a future revision of C++; it encloses a block of expressions that will be evaluated without interference from other threads. The hope is that compilers will be able to leverage hardware implementations of Transactional Memory as they become available, even in commodity hardware [8].

6.5. Outlook

Current C++ implements all basic, mainstream concurrency concepts (except for vectorization, see below); most other languages offer a similar set of features. Some core ingredients, for

instance read / write locks, are still missing; they will likely be part of C++14. Additionally, several new concepts are under investigation for future revisions of C++. There are discussions to provide generic schedulers or implementations thereof such as MapReduce [9], as well as extended fork / join concepts [10]. Resumable functions [11] and `.then` [12, 13] to chain asynchronous calls as known for instance from Java are also considered.

7. Vectorization and C++

Vector instructions operate on a series of input data (SIMD). They leverage current CPU hardware with vector sizes of often four double length floating point numbers, i.e. operating on four double values in one single instruction. General purpose graphic processing units (GPGPUs) offer even larger vector sizes.

7.1. Deployment

C++ does not have an explicit concept of vectorization. It is available as implicit improvement by compilers (similar to optimization) or as language or library extensions, as described in the following. Given the current complexity of deployment, vectorization will likely remain an expert topic at least for the next few years, applied only to "hot code". Several projects have studied the advantages of vectorization in relevant HEP algorithms [14, 15, 16].

7.1.1. Auto-Vectorization For very simple loops, compilers can combine subsequent iterations into vector instructions. This can only happen under stringent conditions, for instance pointer dereferencing and function calls generally prevent auto-vectorization. Code needs to be refactored to match these requirements. Except for the verboseness of the resulting code, no explicit mention exists that the code is optimized for auto-vectorization, making this a fragile state.

7.1.2. Vector Types Instead, input data can be explicitly combined into vectors that overload operators and often mathematical functions as vector operations. As an example, Vc [17] has recently been added to ROOT; a similar proposal was presented to the C++ committee [18].

7.1.3. Intel Cilk Plus SIMD Vectors Instead of relying on an external library one can leverage the compiler's knowledge, all the way through vector instructions. Intel's Cilk Plus SIMD vectors [19] are such a language extension; they implement a new operator syntax `[start:count:stride]` similar to FORTRAN array indexing. This ranged vector access allows the compiler to vectorize the operations – if it sees fit, i.e. this approach leverages the compiler's knowledge about the code and possible optimizations.

7.1.4. Vector Annotation A very traditional approach to vectorization uses language-independent, compiler specific annotations, for instance pragmas [20, 21]. OpenMP 4.0 [22] follows a similar but compiler-independent approach. These annotations usually tell the compiler to change a for-loop's iteration into a vectorized loop, similar to the approach of auto-vectorization, but guaranteeing to the compiler that the code can vectorize, i.e. that it exhibits no inter-iteration dependencies. Their advantage is minimal intrusiveness while providing all necessary information needed for the compiler to vectorize code efficiently.

7.1.5. Language Supported Vectorization Vector annotations change the semantics of for-loops by processing blocks of iterations at once, instead of iterating in sequence. This is a drastic change in memory access semantics. On the other hand, the ability to guarantee to the compiler that a loop *can* be vectorized is becoming more and more relevant. As a consequence, several

proposals exist to make vectorization a language feature [23, 24]. They would be available in all compilers, could leverage the compiler's knowledge, and would explicitly state the vectorization intent. Given that they are based on well known for-loops, they are fairly easy to write and understand.

7.1.6. Tools Function calls can be handled gracefully even in vector loops [24]. But vectorization can include function calls if the function has no side effects, a notion that cannot currently be expressed in C++. There are plans to include the concept of *Elemental Functions* in a future revision of C++.

Mathematical library calls, for instance `exp()`, `sin()` or `sqrt()`, can often prevent vectorization. Implementations exist that circumvent this problem; for instance VDT [25], and this will be integrated into ROOT soon.

7.2. Struct of Arrays (SOA) Memory Layout

Traditional object oriented memory layout favors row-wise storage, where members are stored in memory object by object: $x_0y_0z_0$ $x_1y_1z_1$ $x_2y_2z_2$ $x_3y_3z_3$ for objects 0 to 3. Input data for vector operations must be continuous in memory, favoring a column-wise data layout $x_0x_1x_2x_3$ $y_0y_1y_2y_3$ $z_0z_1z_2z_3$. This latter layout is often referred to as *Struct of Arrays*, as the members of subsequent objects are now stored as arrays. To date, no container exists in C++ that can change a class definition's memory layout (and thus the class definition itself) converting it into an array of column-wise storage. A tool was proposed [26] that facilitates the investigation of structs of arrays, by allowing the collection to transition between traditional memory layout (arrays of structs) and structs of arrays.

8. Conclusions

C++ is a crucial technology used in the development of HEP software. With the standard published in 2011, C++ has changed to a much more concise language. It is much closer to the language we always wanted C++ to be: robust, easy to write, easy to read, yet powerful if needed. It should be exposed to physicists through use in interfaces of frameworks and common tools, such as ROOT and Geant4. As an example and given the current adoption rate of C++11, ROOT plans to migrate to C++11 soon after ROOT 6 has been released. But the evolution of C++ has just started; the delivery process of future C++ revisions has changed to an extent that it will change the way we perceive C++, from a static language to a dynamic one. Many improvements in the C++ language and standard library are on the way; they target increased simplicity, robustness and speed, even of novices' code.

The exposure of current compilers (GCC 4.8, clang 3.3 etc) makes these new features very accessible, whilst offering many improvements such as correctness, performance and dramatically improved diagnostics. It is crucial to expose current compiler and C++ versions also to physicists, maybe even at the cost of a compiler upgrade during the lifetime of a framework version. This can improve C++ for tens of thousands of physicists and reduce the development time for our programs.

References

[1] The C++ Standards Committee `http://www.open-std.org/jtc1/sc22/wg21`
[2] C++ Standard Draft Sources `https://github.com/cplusplus/draft`
[3] Peters A J and Janyst L 2011 Exabyte Scale Storage at CERN *J. Phys.: Conf. Ser.* **331** 052015; Proc. of CHEP2010
[4] Bockelman B P 2014 Big Data - Flexible Data - for HEP *J. Phys.: Conf. Ser.*; Proc. of CHEP2013 (submitted)
[5] Vasilev V *et al* 2012 Cling - The LLVM-based C++ Interpreter *J. Phys.: Conf. Ser.* **396** 052071; Proc. of CHEP2012
[6] Green C *et al* 2012 The art framework *J. Phys.: Conf. Ser.* **396** 022020; Proc. of CHEP2012

[7] Blomer J *et al* 2011 Distributing LHC application software and conditions databases using the CernVM file system *J. Phys.: Conf. Ser.* **331** 042003; Proc. of CHEP2010

[8] Intel Transactional Synchronization Extensions http://www.intel.com/software/tsx

[9] Mysen C *et al* 2013 C++ Mapreduce *JTC1/SC22/WG21* **N3563** http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3563.pdf

[10] Laksberg A and Sutter H 2013 Task Groups As a Lower Level C++ Library Solution To Fork-Join Parallelism *JTC1/SC22/WG21* **N3711** http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3711.pdf

[11] Gustafsson N *et al* 2013 Resumable Functions *JTC1/SC22/WG21* **N3722** http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3722.pdf

[12] Kohlhoff Ch 2013 A Universal Model for Asynchronous Operations *JTC1/SC22/WG21* **N3747** http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3747.pdf

[13] Gustafsson N *et al* 2013 Improvements to `std::future<T>` and Related APIs *JTC1/SC22/WG21* **N3784** http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3784.pdf

[14] Geant Vector Prototype http://geant.cern.ch

[15] Carminati F *et al* 2014 The path toward HEP High Performance Computing *J. Phys.: Conf. Ser.*; Proc. of CHEP2013 (submitted)

[16] Gheata A 2014 Vectorizing the detector geometry to optimize particle transport *J. Phys.: Conf. Ser.*; Proc. of CHEP2013 (submitted)

[17] Kretz M and Lindenstruth V 2012 Vc: A C++ library for explicit vectorization *Softw: Pract. Exper.* **42**: 1409-1430. doi: 10.1002/spe.1149

[18] Kretz M 2013 SIMD Vector Types *JTC1/SC22/WG21* **N3759** http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3759.html

[19] Intel Cilk Plus http://software.intel.com/en-us/intel-cilk-plus

[20] GCC Loop-Specific Pragmas http://gcc.gnu.org/onlinedocs/gcc/Loop_002dSpecific-Pragmas.html

[21] Intel Compiler Documentation: vector pragma http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011Update/compiler_c/cref_cls/common/cppref_pragma_vector.htm

[22] OpenMP 4.0 Specifications http://openmp.org/wp/openmp-specifications

[23] Hoberock J *et al* 2013 A Parallel Algorithms Library *JTC1/SC22/WG21* **N3724** http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3724.pdf

[24] Geva R 2013 Vector Programming: A proposal for WG21 *JTC1/SC22/WG21* **N3734** http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3734.pdf

[25] Piparo D 2014 Speeding up HEP experiments' software with a library of fast and autovectorisable mathematical functions *J. Phys.: Conf. Ser.*; Proc. of CHEP2013 (submitted)

[26] Costanza P 2013 Intel Arrow Street https://github.com/ExaScience/arrow-street