C++ Evolves!

Axel Naumann, CERN
CHEP 2013, Amsterdam

Content

- C++ Language Evolution
- C++ Concurrency Evolution

Content: Language

- The C++ standard and why HEP should participate
- Current work items
- Standards' time to analysis: standardization, compiler, experiments

Content: Concurrency

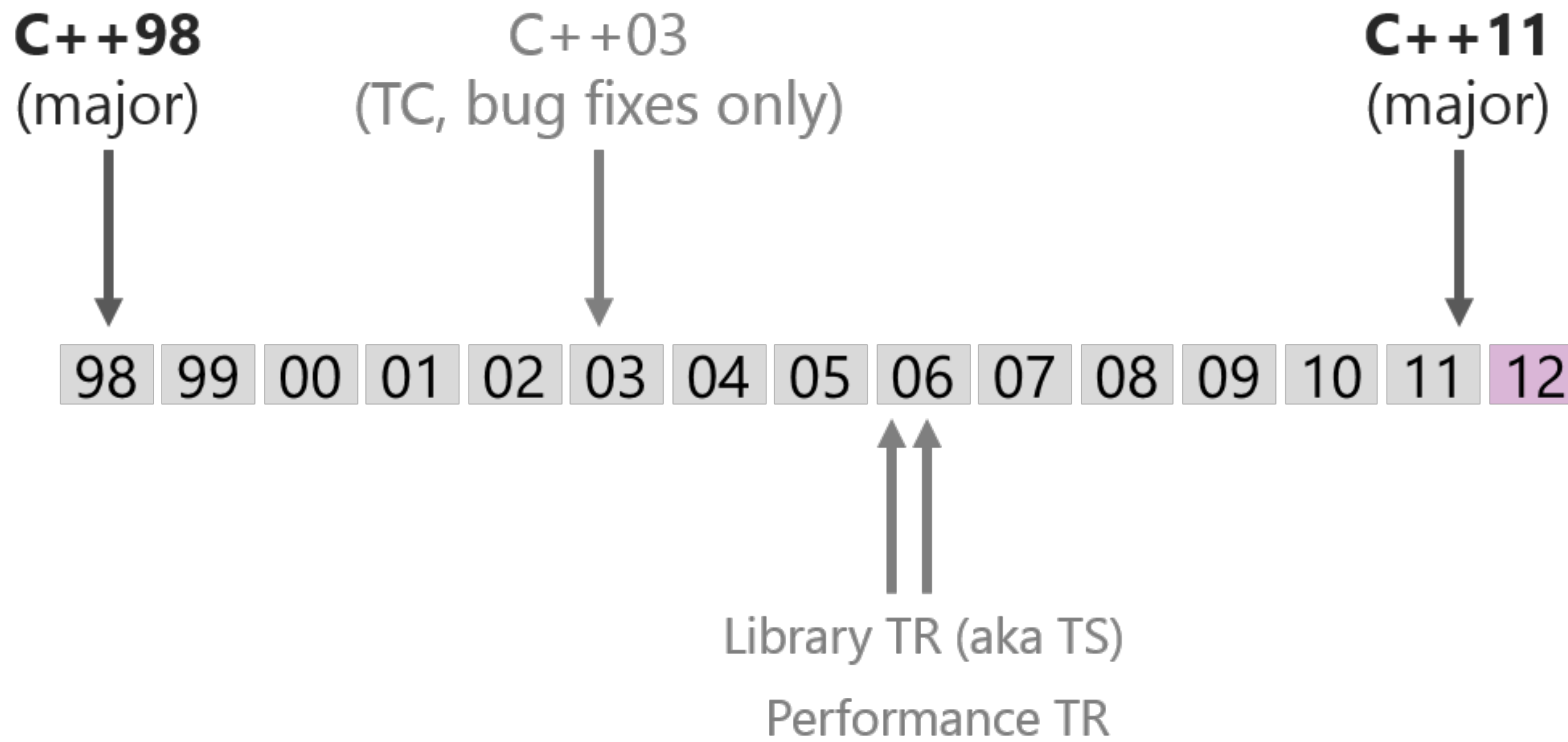
- Glimpse into some other languages
- Concepts of concurrency in C++
- Vectorization

ISO JTC1/SC22/WG21: The C++ Standard

The C++ Standard

- ISO committee, member through national bodies (ANSI, DIN, SNV etc), CERN started around the time Fermilab reduced its participation
    - In principle semi-democratic: all countries have one vote. But technical discussions + “straw polls” involve everyone
- Standards so far: 1998 “C++98” + corrigendum 2003, 2011 “C++11” (formerly known as “C++0x”)

Standardization Past

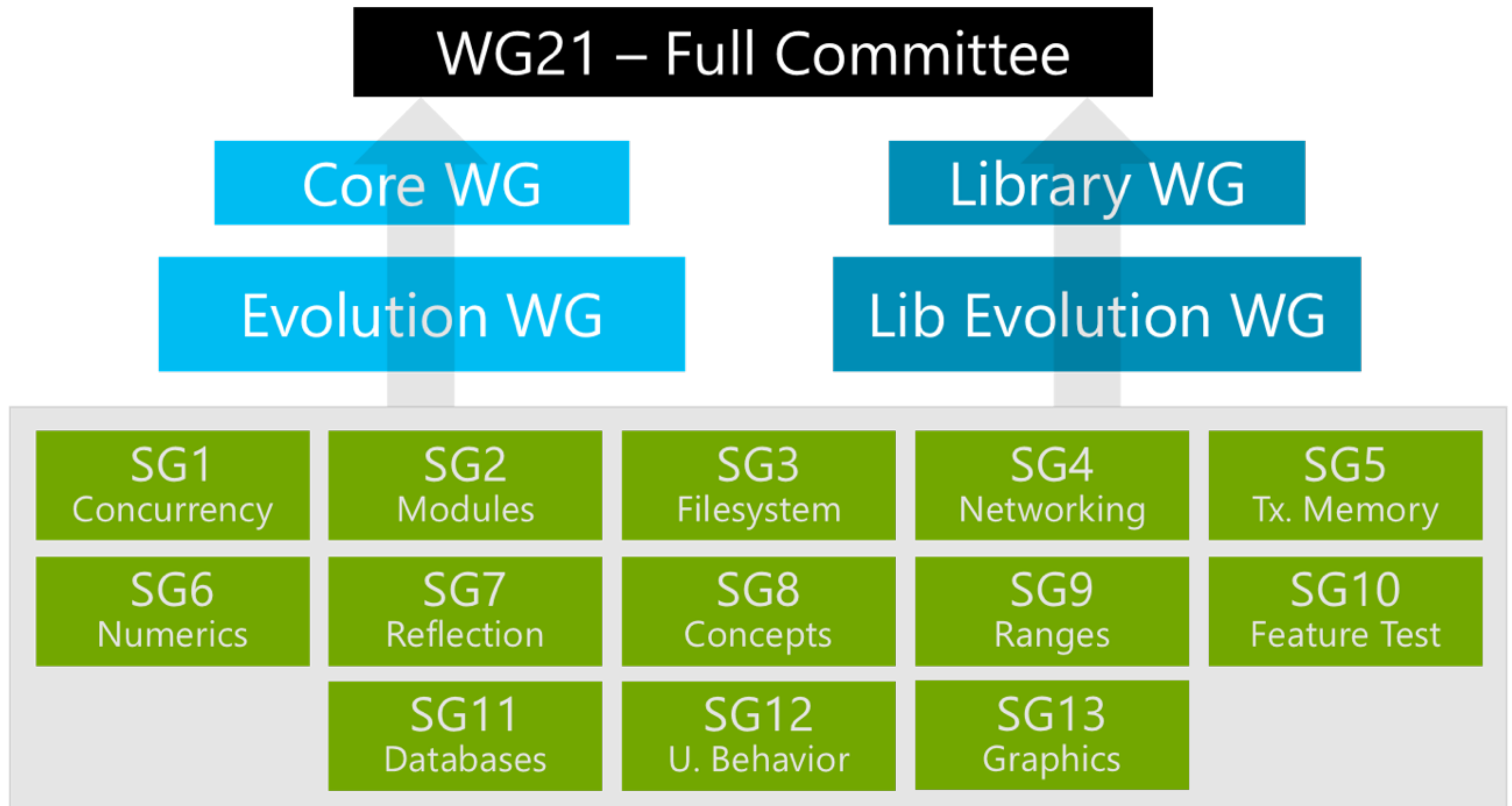


from http://isocpp.org

C++ Standard Plans

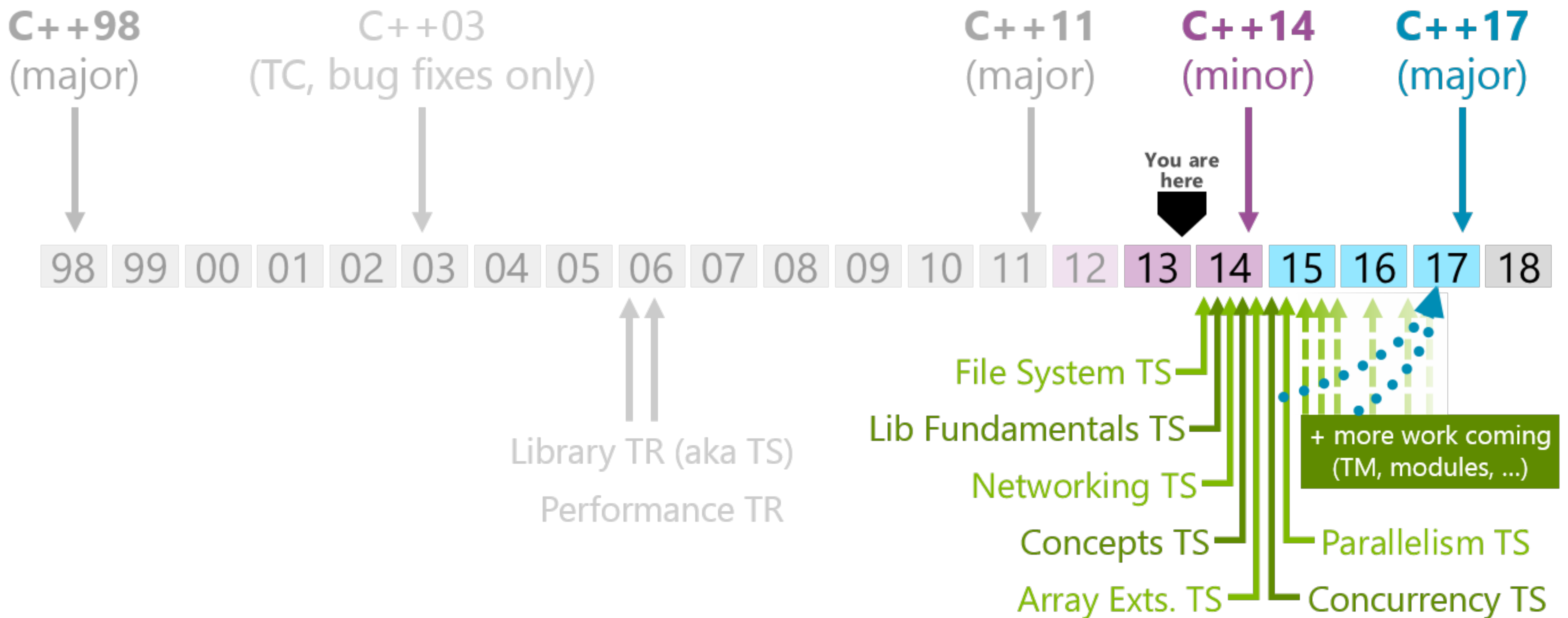
- Lessons from C++11
  - don’t funnel everything into the next standard
  - be careful about intrusive changes
- Faster, smaller updates planned: 2014, 2017
- Parallel work
  - study groups (SG*n*)
  - independent targets (Technical Specifications TS)

Study Groups



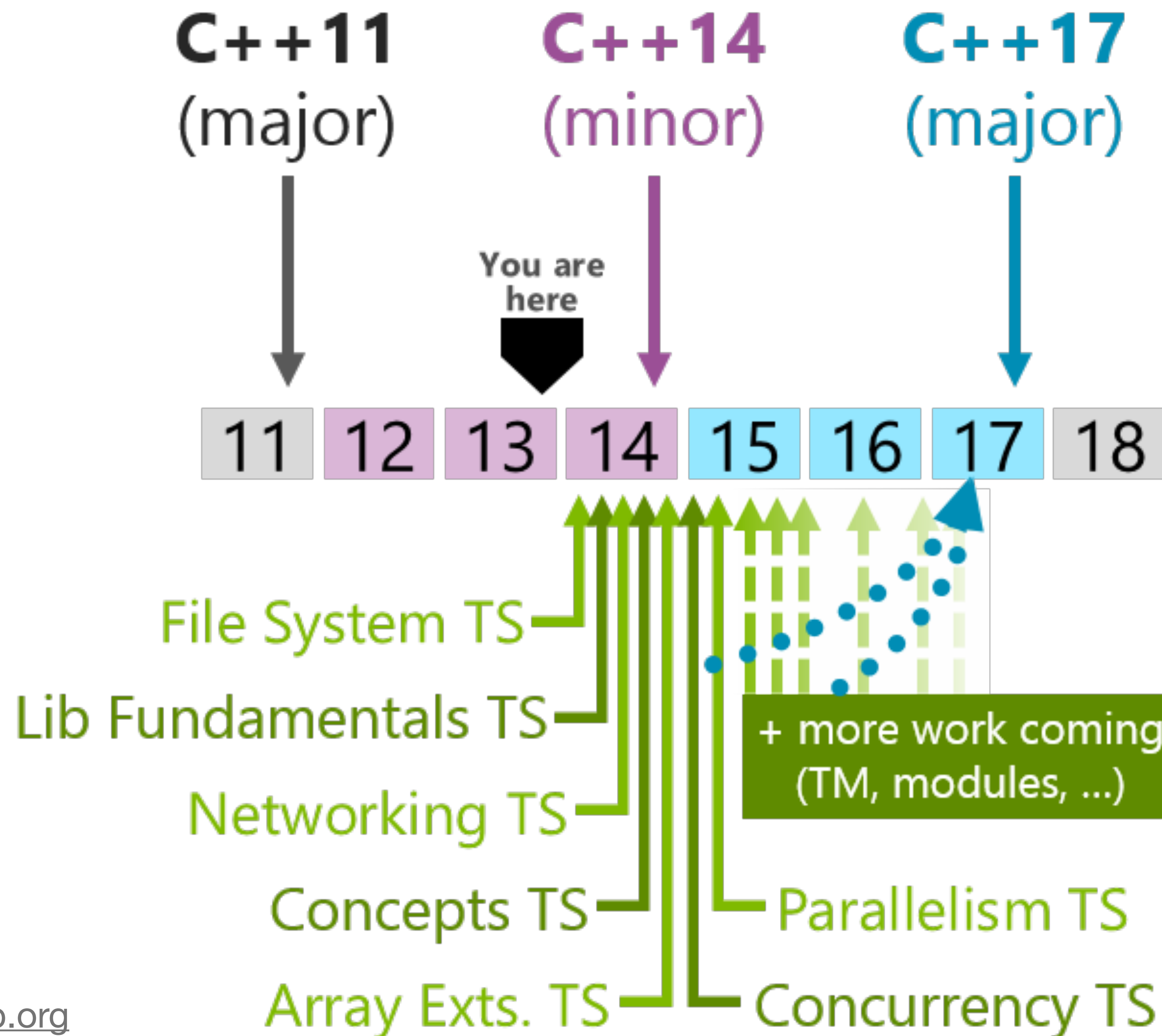
from http://isocpp.org

Current Standardization Plans - Perspective



from http://isocpp.org

Current Standardization Plans



from http://isocpp.org

C++ Committee and HEP

- Committee members: all major IT companies, Boost, computer science
  - users are a scarce resource!
- HEP uses C++ in a unique environment: thousands of developers interacting through headers with high performance code
- Wide knowledge base in production context: is there a Study Group we don't know anything about?

HEP and C++ Committee

- We have invested 10’000 developers * 20 years; we have 50..100 million lines of C++ code
- We educate, study, suffer from and try to get the best out of C++
- We care about C++, we care about its future!
- We are paid by society, we have an obligation to give back what we have learned
- C++ should not be driven (only) by compiler vendors

Introducing C++11, age 2

Relevance of C++11 to HEP

- Bjarne Stroustrup: “C++11 feels like a new language”
- Simpler code
- More expressive code
- Ability to write robust code
- Increased performance

C++11: Performance

- Hashed containers (finally!): std::unordered_map / std::unordered_set to be used instead of e.g. std::map<std::string,...>

- Container initialization

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

C++11: Performance

- Hashed containers (finally!): std::unordered_map / std::unordered_set to be used instead of e.g. std::map<std::string,...>

- Container initialization

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

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

C++11: Performance

- Hashed containers (finally!): std::unordered_map / std::unordered_set to be used instead of e.g. std::map<std::string,...>

- Container initialization

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

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

- Move semantics

```
Collection(Collection&& other);
Collection noCopy = getCollection();
```

C++11: Simple Code

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

C++11: Simple Code

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

- auto

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

C++11: Simple Code

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

- auto

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

- Range-based for

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

C++11: Expressive Code

- Constructor delegation

```
C::C(int i, D* d) { Init(i, d); }
C::C(D* d) { Init(0, d); }
C::C() { Init(-1, 0); }

C::Init(int i, D* d);
```

C++11: Expressive Code

- Constructor delegation

```
C::C(int i, D* d) { Init(i, d); }
C::C(D* d) { Init(0, d); }
C::C() { Init(-1, 0); }

C::Init(int i, D* d);
```

```
C::C(int i, D* d);
C::C(D* d): C(0, d) {}
C::C(): C(-1, 0) {}
```

C++11: Expressive Code

- Constructor delegation

```
C::C(int i, D* d) { Init(i, d); }
C::C(D* d) { Init(0, d); }
C::C() { Init(-1, 0); }

C::Init(int i, D* d);
```

```
C::C(int i, D* d);
C::C(D* d): C(0, d) {}
C::C(): C(-1, 0) {}
```

- Constructor deletion

```
class N {
  // intentionally not implemented
  N(const N&);
```

C++11: Expressive Code

- Constructor delegation

```
C::C(int i, D* d) { Init(i, d); }
C::C(D* d) { Init(0, d); }
C::C() { Init(-1, 0); }

C::Init(int i, D* d);
```

```
C::C(int i, D* d);
C::C(D* d): C(0, d) {}
C::C(): C(-1, 0) {}
```

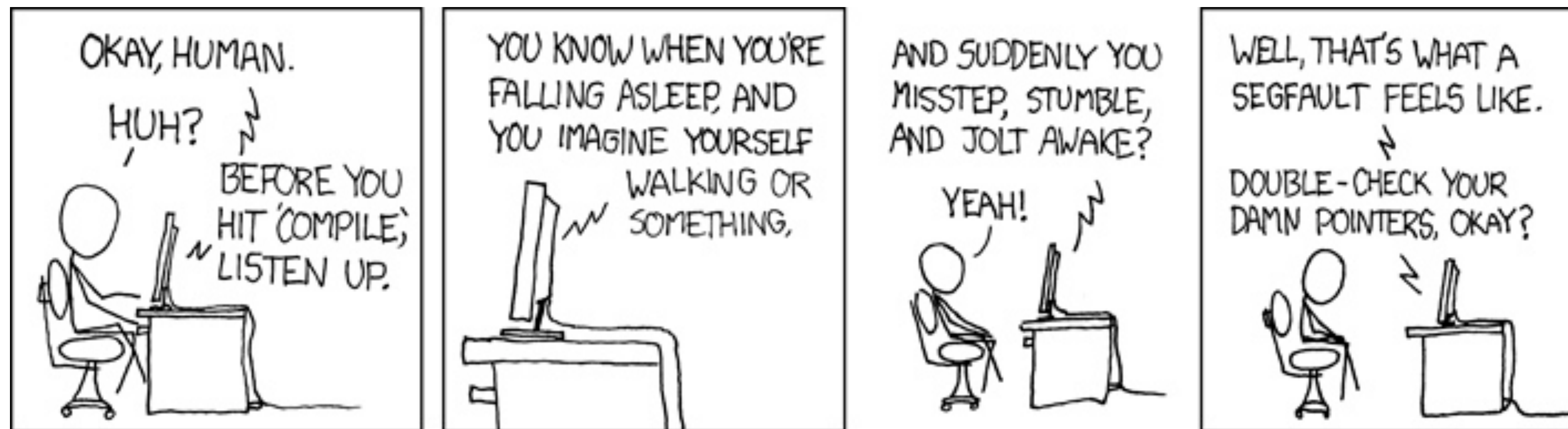
- Constructor deletion

```
class N {
  // intentionally not implemented
  N(const N&);
```

```
class N {
  N(const N&) = delete;
```

C++11: Robust Code

- Pointers versus object ownership



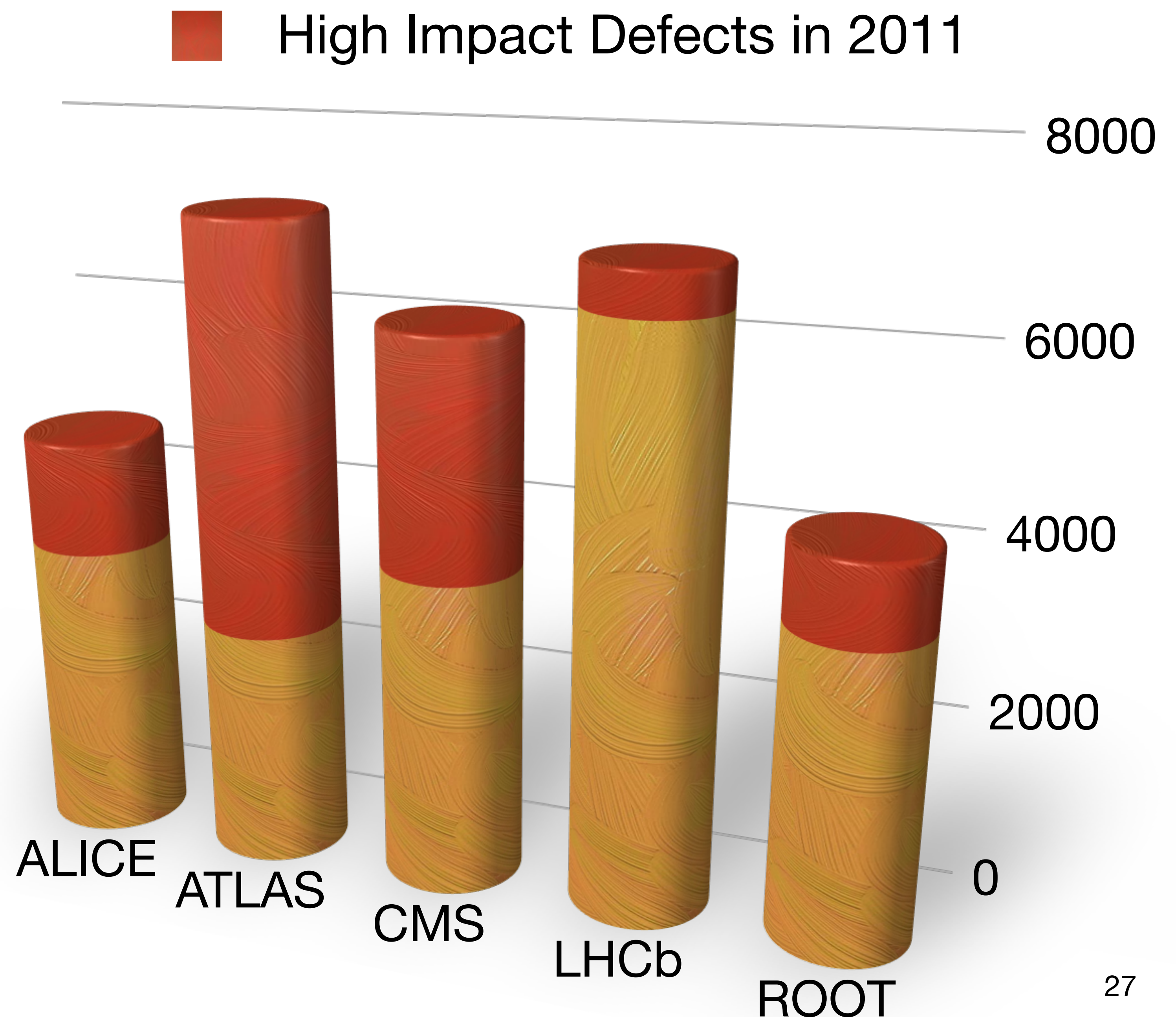
from http://xkcd.com

std::unique_ptr is owned by one, can explicitly pass ownership
std::shared_ptr is reference counted (“garbage collector”)

- Can prevent hours of debugging memory errors!

Code Robustness

- Coverity static analysis: high impact = access after delete etc
- Clear differences in fraction of high impact defects
- Likely caused by coding style / interface definitions
- Demonstrates effect of safe coding - that is accessible to everyone in C++11 [Kowalkowski, Mon 16:07, Effectenbeurszaal]



C++11: The Dark Side?

- Also caters coding wizards
  - variadic templates; lambdas; const_expr; user defined literals;...
- Complex code remains an option in C++11
- Still, C++11 dramatically improves even novices' code

Deploying C++11

From C++03 to C++11

- 100% supported by GCC 4.8, clang 3.3 with flag -std=c++11;
  largely by GCC 4.7, clang 3.2, ICC 14, MSVC 2013

- Old C++ code usually compiles in C++11 “mode”, ROOT had about 8 changes on 3 million lines of code:

  - token#pasting CPP macros

  - x={...} initializers

- Object file compiled with C++11 should not be linked against old C++:
  all C++11 or none

Current Adoption

- Few polls show a wide range of usage:
  - NOvA physicists use C++11
  - CMS, ATLAS, LHCb, Belle II are using it in frameworks and reconstruction
  - ATLAS, LHCb: offering it for physics analyses in near future
  - FairRoot, ALICE validated to compile in C++11
  - RHIC uses C++0x as available in GCC <= 4.6; move to C++11 in 2015
- Analysis usage (except for NOvA): approximately 0%

Reasons for Non-Adoption

- Physicists: Do you use C++11?
  - “I have no idea what C++11 is.”
  - “I had to google it.”
- Frameworks:
  - Perceived lack of incentives
  - Production compiler versions: SL5/6

Tracking Current Compilers

- Several experiments decided to aggressively follow newest compilers, enabling the use of C++11 almost as a side effect:
  - CMS’s Peter Elmer: “the compiler clearly has improved code generation (i.e. performance), auto-vectorization has improved, the code parser is better, etc.”
  - Belle II’s Thomas Kuhr: “It can help to attract new students [...]. C++11 has very nice features and it's bad for the motivation and education of developers if one has to tell them that they cannot use these features.”
- Imminent streamlined delivery process of C++ standard / library calls for early adoption of compilers

Deployment of Current C++

- Modern compilers solves frequent user complaint: diagnostics!

```
std::find(vec.begin(), ConstVec.end(), 12);
```

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

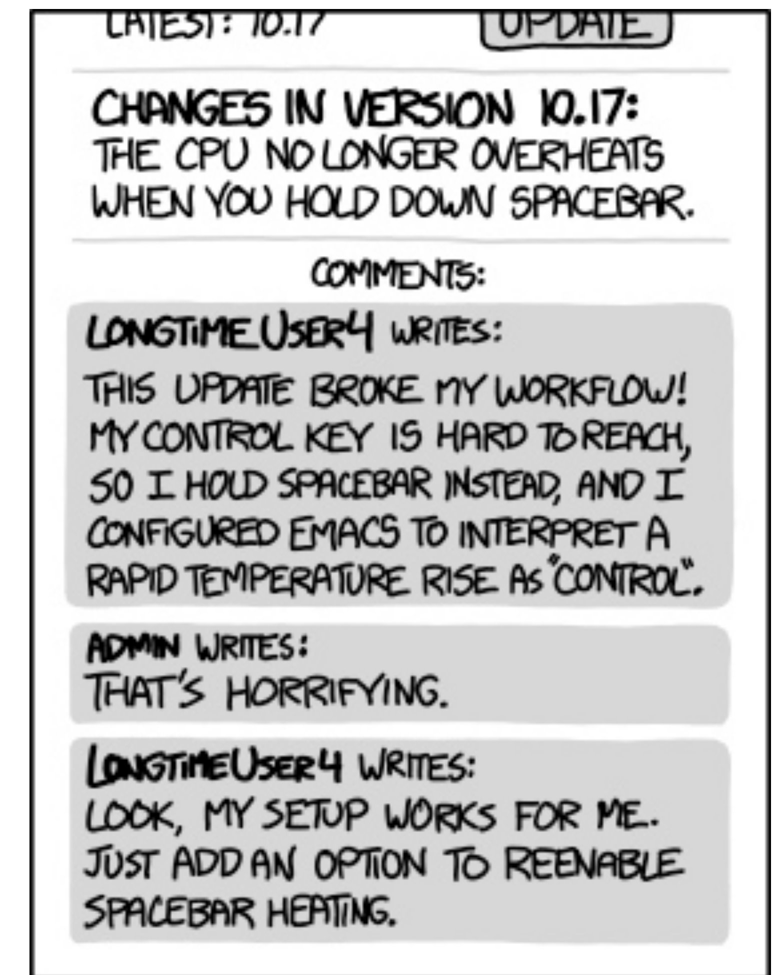
C++ Deployment

- Modern compilers solves frequent user complaint: diagnostics!

```
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.
```

Language Summary

- The language has changed dramatically
- Many benefits especially for casual coders: safe, simple, expressive code
    - ownership clarification
    - concise constructs for common idioms
- Improved standard library
- It saves time!



from http://xkcd.com

Personal Concurrency Survey

Each Language Has Its Own Philosophy

- Disclaimer: there are many wonderful languages. You should use them all.
- Will discuss random subset, showing specific concepts coupled to the language design
- Code snippets are fragments, demonstrating concurrency
- Meant to give a taste of concurrency in (modern) languages

Verilog [1984]

- Event-driven hardware description language; blocks of code connected through data flow

```
event e;
initial
 repeat(4)
 begin
   #20; // wait 20 time units
   ->e ; // signal e
   $display("e triggered");
 end
always
 begin
   #10;
   if(e.triggered)
     $terminate;
   end
```

Haskell [1990]

- Parallelism through “sparks”: tasks scheduled by the runtime (“par”)

```
import Control.Parallel

nfib :: Int -> Int
nfib n | n <= 1 = 1
       | otherwise = par n1 (pseq n2 (n1 + n2 + 1))
                     where n1 = nfib (n-1)
                           n2 = nfib (n-2)
```

D [2001]

- Threads, message passing

```
void spawnedFunc(Tid tid) {
    receive(
        (int i) { writeln("Received ", i);}
    );
    // Send a message back to the owner thread
    send(tid, true);
}
void main() {
    // Start spawnedFunc in a new thread.
    auto tid = spawn(&spawnedFunc, thisTid);
    send(tid, 42);
    auto wasSuccessful = receiveOnly!(bool);
}
```

Scala [2003]

- Primarily based on actors: do something, then tell someone

```
class Pong extends Actor {
  def act() {
    while (true) {
      receive {
        case Ping =>
          Console.println("Pong")
          sender ! Pong
        case Stop =>
          Console.println("Pong: stop")
          exit()
...
```

Scala (2)

- Messages and futures: handle future result

```
val future = new FutureTask[String](new Callable[String]() {
  def call(): String = {
    searcher.search(target);
}})
executor.execute(future)
```

Block until the result is available

```
val blockingResult = future.get()
```

Clojure [2007]

- Immutable states, parallel transitions

```
(defn test-stm [nitems nthreads niters]
  (let [refs  (map ref (repeat nitems 0))
        pool  (Executors/newFixedThreadPool nthreads)
        tasks (map (fn [t]
                      (fn []
                        (dotimes [n niters]
                          (dosync
                            (doseq [r refs]
                              (alter r + 1 t))))))
                   (range nthreads))]
    (doseq [future (.invokeAll pool tasks)]
      (.get future))
    (.shutdown pool)
    (map deref refs)))
```

Go [2009]

- Goroutines: light-weight threads

```
go someFunction()
```

- Channels: communication, synchronization

```
c := make(chan int)  // Allocate a channel.
// Start the sort in a goroutine
go func() {
    list.Sort()
    c <- 1  // Send signal on completion.
}()
doSomethingForAWhile()
<-c   // Wait for sort to finish.
```

[Binet, Mon 15, poster]

Concurrency In C++11

Threads, Synchronization Mechanisms

- C++11 provides wrapper around OS threads, replacing pthreads / Windows threads for basic interaction

```
auto myThread = std::thread(workerFcn);
doSomething();
myThread.join();
```

- C++11 provides synchronization primitives: std::mutex, std::lock_guard, std::condition_variable and more

- All fundamental threading ingredients in C++11

Tasks

- Let the runtime schedule a task
- Might not even start a separate thread at all
- Could be light-weight and / or pooled thread

```
auto task = std::async(workerFcn);
doSomething();
task.get();
```

Futures

- A handle of a value that does not need to exist yet
- Delays evaluation as much as possible

```
future<int> fut = std::async(func);
int result = thisTakesAWhile();
result *= fut.get();
```

Memory Synchronization

- Thread local variables with one copy per thread

```
thread_local int i;
```

- Atomics where certain operations are protected from race conditions

```
atomic_int i;
void threadedWorker() {
  i += 42;
}
```

- Transactional Memory expected as future Technical Specification. Provides all-or-nothing blocks (transactions)

```
__transaction_atomic { if (x < 10) y++; }
```

C++ Parallelization Outlook

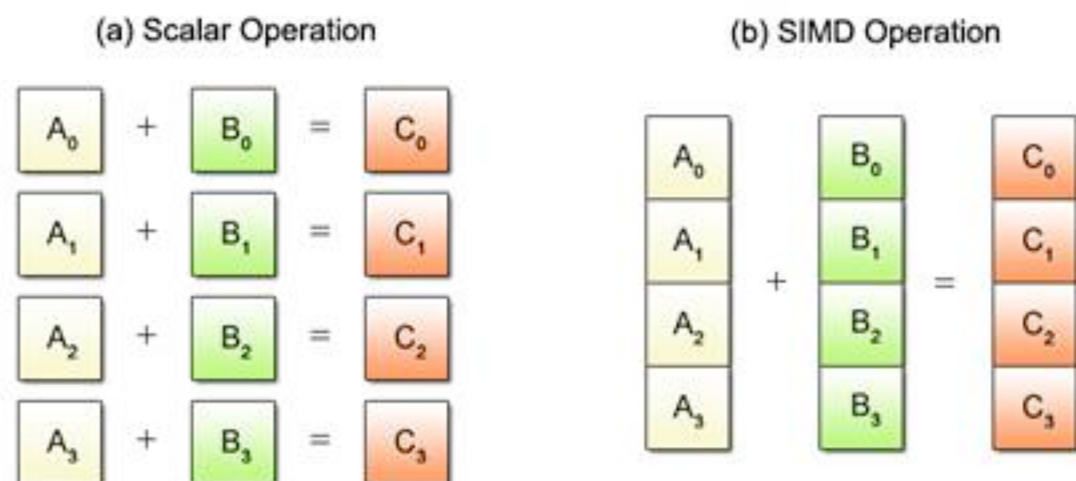
- Extended fork / join concepts, e.g. task groups
- .then
- Resumable functions
- Schedulers / MapReduce
- Read / write locks

Bringing Vectorization into C++

Vectorization

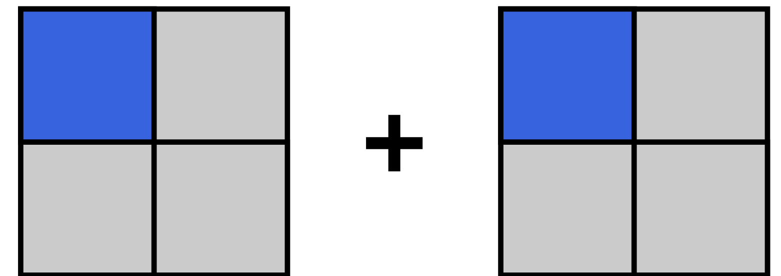
- Traditional a **+** b operation: combine input a with input b; yields one output
  Single Instruction, single data

- Current CPUs run **+** operation on one or multiple inputs (consecutive in memory) at the same cost, e.g. SSE2 instructions
  Single Instruction, Multiple Data (SIMD)



Vectorization Versus Performance

- We often use only one “slot” out of four
  - likely even more dramatic in the future
  - already more dramatic for GPUs
- Vectorized code commonly sees throughput increase by factor 2
  - also due to different data / code access patterns, caching effects



Auto-Vectorization

- Compilers combine simple loop iterations into vector operations
- Function call, pointer access etc prevent auto-vectorization
- Advantage: needs no extra code; leverages compiler knowledge and optimization
- Disadvantage: rarely possible; needs intrusive code refactoring; gets easily broken also because the vectorization is not explicitly visible (except for "ugliness" of code)

Vector Types

- Can use new type: vector of input values
- With explicit vector CPU instructions (intrinsics), for instance by overloading operator+()
- Example: Vc, now in ROOT 6, soon in ROOT 5.34; proposed to C++ N3571

```
Vc::float_v x = ...; Vc::float_v y = ...;
Vc::float_v r = Vc::Sqrt(x*x + y*y);
```

- Advantage: high performance gains; explicit vectorization; code usually "makes sense" (i.e. human brain is okay with vector operations)
- Disadvantage: needs code refactoring; depends on CPU specifics (but hidden in library); interferes with compiler's work

Intel Cilk Plus SIMD Vectors

- New array notation with index ranges and stride [from:num:stride]

```
a[:] = 5; // set all elements to 5
a[0:7] = 17; // set first 7 elements to 17
a[1:4:2] = 17; // set first 4 odd elements (“stride 2”) to 1
```

- Defines operations

```
a[:] = b[:] + 5;
c[:] = a[:] + b[:];
```

- Works also for multiple dimensions

```
d[:][:] = 12;
```

- Disadvantages: language extension; limited compiler support (ICC; GCC + clang not there yet)

Vector Annotation

- For instance OpenMP 4.0 or compiler specific pragmas

```
void add_floats(float *a, float *b, int n){
  int i;
#pragma simd
  for (i=0; i<n; i++){
    a[i] = a[i] + b[i];
  }
}
```

- Advantage: long history; language agnostic; simple to write because based on well understood loops

- Disadvantage: lots of tweaks needed; architecture specific; changes loop semantics (iteration sequence) without being part of the language

Language Supported Vectorization

- Two C++ proposals discussed
  - std::for_each(std::vec, what_to_do); N3554
  - simd_for(auto i: collection) {...}; N3561
- Latter clearly signals different loop semantics
- Disadvantage: more pages in the standard!
- Advantages: standard (compiler support), leverages compiler knowledge; based on for loops thus easy to write / understand

Elemental Functions; Vectorized Math

- simd_for gracefully handles calls to non-vectorized functions
- Vectorization benefits from new concept *elemental* function: “function that does not have side effects”
  - currently compiler-specific annotation, will likely end up in C++
- Math functions should be vectorization-friendly; see e.g. VDT https://svnweb.cern.ch/trac/vdt [Piparo, Tue 17:47, Effectenbeurszaal]

Vectorization in Practice

- Vectorized code must
  - not use virtual functions
  - align data as vectors
- Very intrusive
  - changes interfaces
  - changes data formats
- But it gives you a very noticeable performance boost already today

Vectorization Efforts

- Several lessons already learned
  - Vectorization improves cache locality
  - Refactoring is non-trivial
- Example presentations at CHEP 2013:
  - Geant V [Carminati, Thu 12:06, Effectenbeurszaal]
  - Vectorized geometry [Gheata, Mon 17:46, Effectenbeurszaal]

Structs Of Arrays

C++ Memory Layout, Or The Curse Of Object Oriented Data

```
class XYZ {
  double x;
  double y;
  double z;
};
```

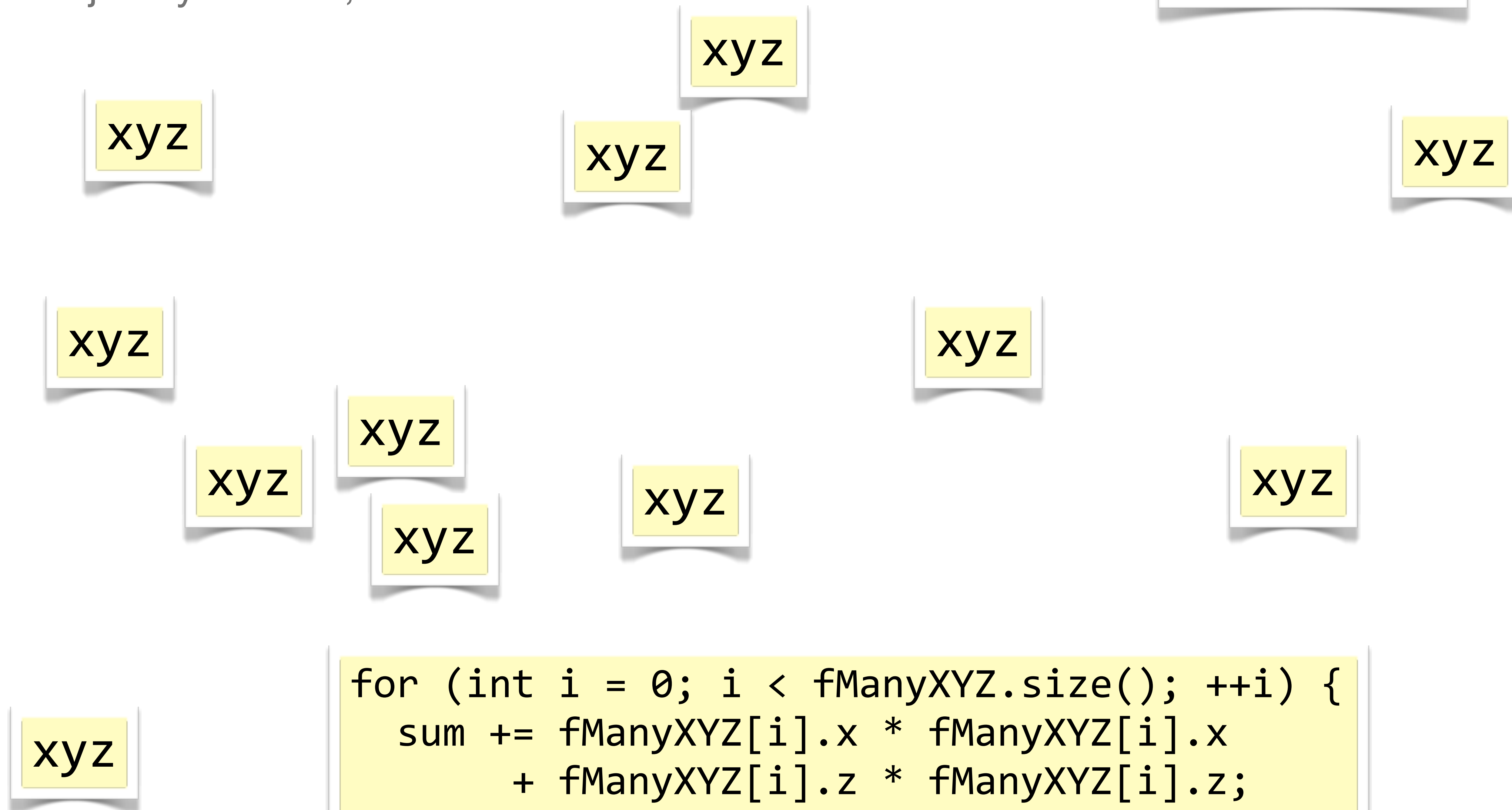
- Assume algorithm that calculates

```
for (int i = 0; i < fManyXYZ.size(); ++i) {
  sum += fManyXYZ[i].x * fManyXYZ[i].x
       + fManyXYZ[i].z * fManyXYZ[i].z;
}
```

Array of Pointers to Structs

```
class XYZ {
   double x;
   double y;
   double z;
};
```

- “TObjArray<XYZ>”, vector<XYZ*>



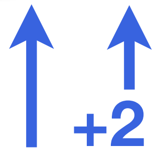

```
for (int i = 0; i < fManyXYZ.size(); ++i) {
  sum += fManyXYZ[i].x * fManyXYZ[i].x
       + fManyXYZ[i].z * fManyXYZ[i].z;
}
```

Array of Structs

```
class XYZ {
  double x;
  double y;
  double z;
};
```

- vector<XYZ>, XYZ[N]

```
xyz xyz xyz xyz xyz xyz xyz xyz xyz yxz
```

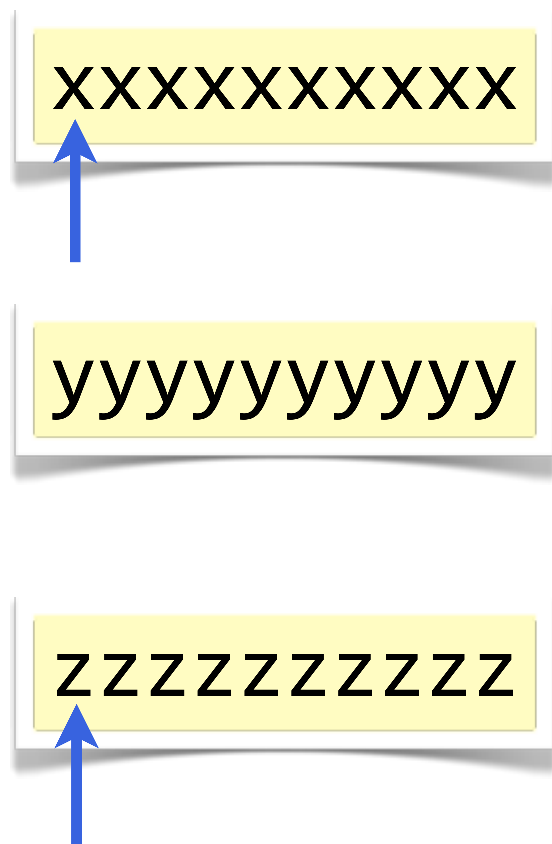



```
for (int i = 0; i < fManyXYZ.size(); ++i) {
  sum += fManyXYZ[i].x * fManyXYZ[i].x
       + fManyXYZ[i].z * fManyXYZ[i].z;
}
```

Structs of Arrays (SOA)

```
class XYZ {
   double x;
   double y;
   double z;
};
```

- structOfArray<XYZ>




```
for (int i = 0; i < fManyXYZ.size(); ++i) {
  sum += fManyXYZ[i].x * fManyXYZ[i].x
       + fManyXYZ[i].z * fManyXYZ[i].z;
}
```

SOA Memory Layout

```
class XYZ {
   double x;
   double y;
   double z;
};
```

- structOfArray<XYZ>

```
xxxxxxxxxx
```

SSE

```
yyyyyyyyyy
```

```
zzzzzzzzzz
```

SSE

```
for (int i = 0; i < fManyXYZ.size(); ++i) {
   sum += fManyXYZ[i].x * fManyXYZ[i].x
        + fManyXYZ[i].z * fManyXYZ[i].z;
}
```

SOA Element Access

- Accessing element of object number i

  what used to be `fManyXYZ[i].x`

  now becomes `fManyXYZ.x[i]`

```
NaiveXYZSOA {
  vector<double> x;
  vector<double> y;
  vector<double> z;
} fManyXYZ;
```

- Workaround, wonderful R&D tool: Intel's Arrow Street [Costanza, Mon 17:25, Effectenbeurszaal]

- Proper solution to convert vector<Jets> into struct of arrays:
  - language support, or
  - library component, but needs C++ reflection (element description)

Concurrency Summary

- All basic ingredients in C++11, even more to come
- Standard library instead of custom implementations (e.g. ROOT TThread)
- Available concepts of concurrency match those of other languages, no intrinsic disadvantage of using C++
- More concepts will become available - your feedback is welcome!

Conclusion

For Experiments

- Don't convert to a new language standard, but prepare for a continuous standard and compiler delivery process
- Benefit from safer C++
- Benefit from better compilers

For Physicists

- There is nothing to be done *by them* - they should not need to act
- ROOT, Geant, frameworks should demonstrate the advantage of simple code, clear ownership, improved standard library

```
TH1::AddFunction(std::unique_ptr<TF1>)
```

- C++11 and after brings us closer to the ultimate goal:
  - Write correct code and analyses easily!
  - From data taking to physics result quickly!