Science & Technology Facilities Council
Rutherford Appleton Laboratory

TTreeIterator
an STL-friendly TTree API

[updated 9th June 2021]

Tim Adye
Rutherford Appleton Laboratory

originally presented at
ROOT Parallelism, Performance and Programming Model Meeting
27th May 2021

Outline

- Critique of existing TTree read access methods
  1. SetBranchAddress
  2. TTreeReader
  - will not discuss RDataFrame (or TTree::Draw)
    - I have no complaints!
      - I personally prefer functional-style code, and RDataFrame is the way to go
    - however I still think there is a place where a loop-over-entries can be more convenient, eg.
      a. filling histograms with information computed from many columns
      b. transforming entries into another format
      c. for users more familiar with loops
- TTreeIterator
  - just a personal project, exploring some ideas to improve TTree reading/filling
    - Developed in a branch of another development project (RooFitTrees)
      - works well, but currently undocumented and in need of code cleanup
    - I hope maybe some of the ideas can be useful more widely, even if not adopted wholesale
  3. API choices
  4. interesting aspects of the C++11 implementation
  5. Performance measurements
  6. optimisation… optimisation… optimisation

Traditional way to read an n-tuple

- Loop over entries, like this:

```
TTree* tree= file.Get<TTree>("xyz");

double vx, vy, vz;
tree->SetBranchAddress("vx",&vx);
tree->SetBranchAddress("vy",&vx);
tree->SetBranchAddress("vz",&vz);

Long64_t n = tree->GetEntries();
for (Long64_t i=0; i<n; i++) {
  tree->GetEntry(i);
  hxy.Fill (vx, vy);
  hz .Fill (vz);
}

tree->ResetBranchAddresses();
delete tree;
```

spooky action at a distance, “side effects”

- This example just fills histograms `hxy` and `hz`, which would be easier done with `tree->Draw()`
  - a real application of an event loop would perform calculations on many variables and fill histograms with derived quantities

- The use of `SetBranchAddress` is cumbersome, fragile, and error-prone, eg.
  - likely segfault if we access the tree again in another routine without `ResetBranchAddresses`
  - need `*value`, but `**object`
  - errors can easily go unnoticed
    - eg. did you spot the error? Will silently use the wrong values for `vx` and `vy`.
    - Each variable specified 3 times
    - This mistake happens more often when variable definition and branch association is separated from use
  - amount of boiler-plate scales with the number of (10-100s) variables
  - trouble sharing variables with functions called inside loop
  - C-style, not C++

TTreeReader is much better

- TTreeReader is a more C++ish alternative

```
TTreeReader tree("xyz", &file);
TTreeReaderValue<double> vx (tree, "vx");
TTreeReaderValue<double> vy (tree, "vx");
TTreeReaderValue<double> vz (tree, "vz");

while (tree.Next()) {
  hxy.Fill (*vx, *vy);
  hz .Fill (*vz);
}
```

side effects at least more clearly visible

- (can also use an iterator, but dereferencing the iterator doesn't do anything useful.)

- Unfortunately I have yet to see TTreeReader used in physics code
  - always one of `TTree::Draw`, `SetBranchAddress`, or more recently RDataFrame
  - perhaps TTreeReader needs better advertisement
    - brief mention in ROOT user manual

- Still some of the same problems with TTreeReader, just less so
  - Harder to make a mistake, but…
    - eg. this error will silently use the wrong values for `vy`.
    - Each variable now specified 2 times
    - The variable definition and branch association is still separated from use
  - amount of boiler-plate scales with the number of variables
  - sharing variables with functions called inside loop even harder
    - can't make `vx`, `vy`, `vz` global variables
  - C++98 style

TTreeIterator – another way to read a TTree

- TTreeIterator is inspired by PyROOT's iterator interface for TTrees
  - Implemented an interface, a bit like `std::vector< std::map< std::string, std::any > > >`
  - C++ requires types and methods to be defined at compile-time, so it's a little more complicated.

PyROOT

```
tree = file.Get("xyz")
for entry in tree:
    hxy.Fill (entry.vx, entry.vy)
    hz .Fill (entry.vz)
```

TTree `"xyz"` read from file `file` (default=current file) – like TTreeReader

Range-for loop over all entries in the tree

```
TTreeIterator tree("xyz", &file);
for (auto& entry : tree) {
  hxy.Fill (entry.Get<double>("vx"),
            entry.Get<double>("vy"));
  hz .Fill (entry["vz"]);
}
```

Values to read defined only once, when needed (no visible side-effects)

If type is unambiguously known from context, can use map-style syntax

- Improved error checking
  - compile-time check for ambiguous or incompatible type
    - eg. 2D `TH2D::Fill` can take other than just `double`s, so need to specify explicitly – or load into a local variable
  - run-time check for missing branch or wrong type
    - gives error message and returns `NaN` (or user-specified value) – maybe better to throw exception

TTreeIterator features

- TTreeIterator works with any data type that can be saved in a TTree, eg.
  - `std::vector`, `std::string`, eg.

```
const std::vector<double>& vec = entry["vec"];
const std::string&         str = entry["str"];
```

  - `TObjects` like `TH1D`, `TRandom`, `TUUID`, eg.

```
const TH1D& hist = entry["hist"];
```

  - C-style `struct`, eg.

```
struct MyStruct {
  double x[3];
  int i;
};
const MyStruct& M = entry["M"];
double z = M.x[2];
```

- Can be used in combination with the other access techniques, eg.
  - TTree created or read from file by the user
  - traditional `for` loop + `tree.GetEntry(i)`
  - `SetBranchAddress` for same or different variables

Accessing data values

- Here are some example accesses:

```
auto   x = entry.Get<double>("x");
double y = entry[“y“];
const std::vector<double>& vec = entry["vec"];
const std::string&         str = entry["str"];
```

- the map-style accessor can be used as long as the type can be determined unambiguously at compile-time
  - `Get<double>("x")` can always be used to specify type explicitly
    - would it be better to always do this?
- here we access the doubles by value, but the vector and string by reference
  - this is more efficient for the larger types, but both styles work

STL algorithms

- TTreeIterator::iterator conforms to the iterator requirements
  - it can be used in STL algorithms, eg.

```
auto sum = std::accumulate (tree.begin(), tree.end(), 0.0,
                            [](double s, const TTreeIterator& entry) {
                              return s + entry.Get<double>("x");
                            });
```

  - or

```
std::vector<double> vx;
std::transform (tree.begin(), tree.end(), std::back_inserter(vx),
                [](const TTreeIterator& entry) -> double { return entry["x"]; });
```

- This type of access is more suited to RDataFrame, so I haven’t pursued it further
  - I don’t know if this would work with STL algorithms’ parallel execution
    - likely problems with shared access to underlying tree

Filling with TTreeIterator

- TTreeIterator can also be used to fill a TTree, eg.

```
TTreeIterator tree("xyz", &file);
for (auto& entry : tree.FillEntries(10000)) {
  entry["vx"] = gRandom->Gaus(2,3);
  entry["vy"] = gRandom->Gaus(-1,2);
  entry["vz"] = gRandom->Gaus(0,100);
  entry.Fill();
}
```

- No “TTreeWriter” to improve over `TTree::Branch()`
  - Advantages of TTreeIterator even clearer for filling
- Branches created when first set
  - automatic catch-up if not defined on first iteration
  - fills unset values with `NaN` (or user-defined value), not the last-set value
- No call to `Write()` required
  - Leave `Fill()` to user, so can skip entry with `continue`
- Outside of simple examples like this, range for-loop may be less useful
  - When filling in a loop over input data, use `auto ientry=tree.FillEntries()` and explicit `ientry->Fill()`
  - would it help to use a `std::back_inserter`-style model for filling?

Data types with Fill

- Data type can usually be inferred when setting value, so simple map-style syntax is usually sufficient here
  - Checks for incompatible types at run-time (eg. when appending to an existing tree)
- Filling also works for all types, eg.

```
entry["str"] = std::string("value");
```

  - C-style `struct` leaf types can be defined when set or, more conveniently, at compile-time in the `struct`, eg.

```
struct MyStruct {
  double x[3];
  int i;
  constexpr static const char* leaflist = "x[3]/D:i/I";
};
entry["M"] = MyStruct{1.0, 2.3, 4.9, 6};
```

Implementation details: std::any

- For our example accesses:

```
for (auto& entry : tree) {
  auto   x = entry.Get<double>("x");
  double y = entry[“y“];
  const std::vector<double>& vec = entry["vec"];
  const std::string&         str = entry["str"];
}
```

- internally, TTreeIterator keeps a cache of each value, keyed on
  - `std::pair(branch_name, data_type)`
- on first use, calls `SetBranchAddress` with the address of the cached value stored in an `std::any`-like object
  - `std::any` is a convenient C++ standard way to perform type-erasure
    - works for basic or complex types
    - for small types (like `double`), doesn't require an extra `new`
  - unfortunately, `std::any` was only added in C++17
  - fortunately it as a header-only library in GCC, so I
    - borrowed the implementation
    - tidied up the code – doesn't need to conform to STL internals
    - renamed it to `Cpp11::any` so it doesn't clash
    - found a significant optimisation, so I use this version by default even in C++17
    - Provided patch to GCC libstdc++, accepted 2021-06-04

Implementation details: type deduction

- For our example accesses:

```
double y = entry[“y“];
const std::vector<double>& vec = entry["vec"];
const std::string&         str = entry["str"];
```

  - The type of `entry["name"]` has to be determined at compile time
  - implemented like this:

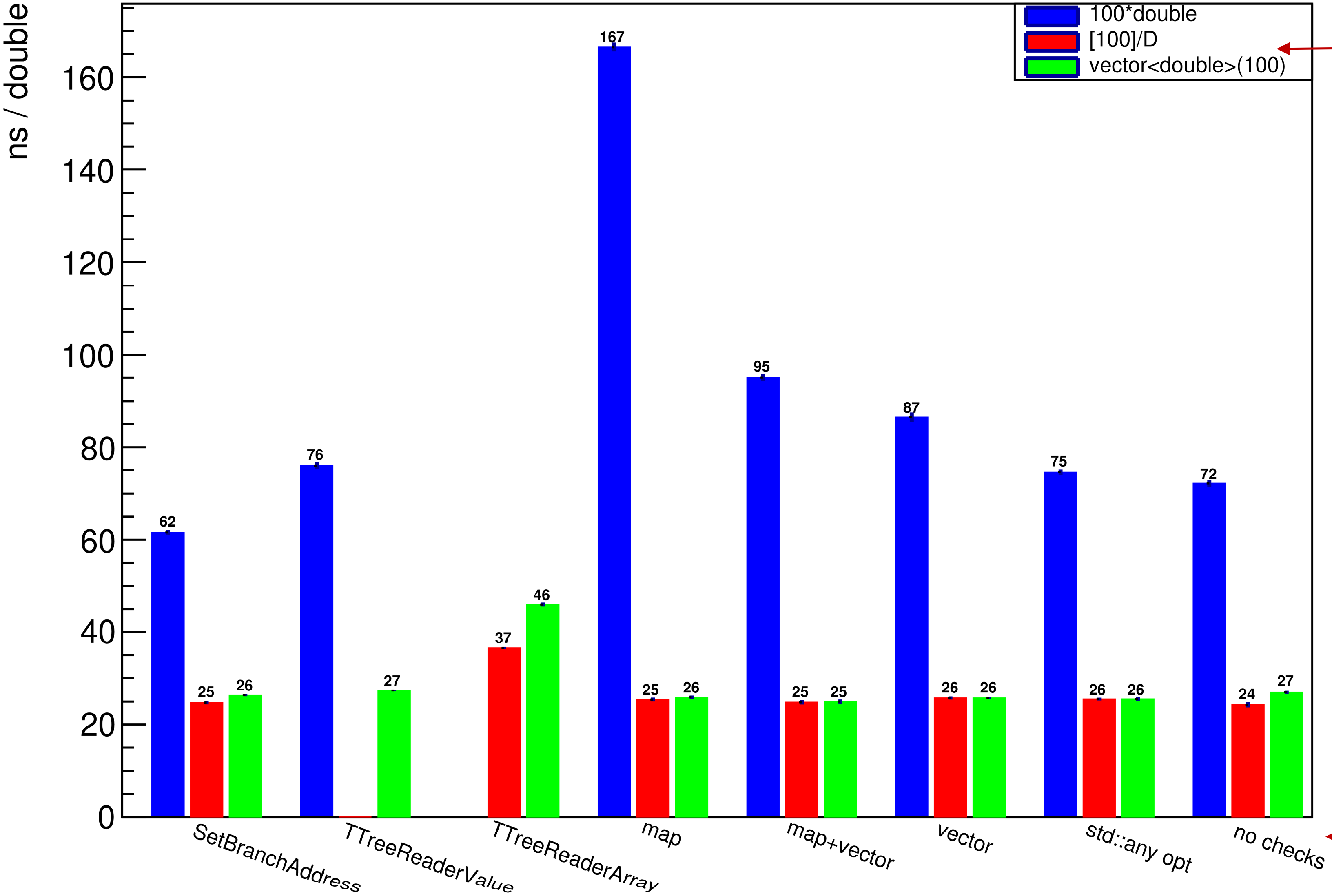
```
Getter operator[] (const char* name) const { return Getter(*this,name); }

struct Getter {
  Getter(const TTreeIterator& entry, const char* name) : fEntry(entry), fName(name) {}
  template <typename T> operator const T&() const { return fEntry.Get<T>(fName); }

  const TTreeIterator& fEntry;
  const char* fName;
};
```

  - all these shenanigans can be optimised away by the compiler
  - the use of the `operator const T&() const` is the magic here that simulates overloading on the return type
    - I haven’t seen much discussion on this method, but it works nicely in many cases
    - need to specify type explicitly (`double(entry["x“])` or `entry.Get<double>("x")`) if used in expressions or with overloaded function calls
      - flagged at compilation, so not a source of error
- Could be improved. Or is this map-style access feature worth the complication?

Performance measurements



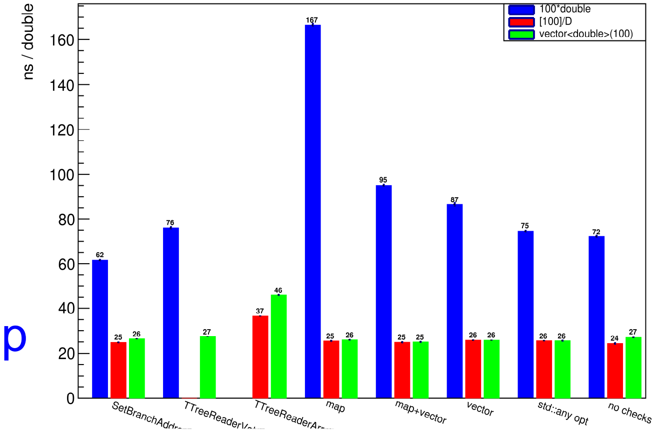
data types accessed

access method or optimisation

- Each test sums 100 doubles in 500,000 entries
- Each test repeated 10 times for ~0.5ns uncertainty on average

- Run on empty Xeon E5-2620v4 2.1GHz (8 cores, HT)
- CentOS 7.9.2009, ROOT 6.24/00, GCC 10.1.0, C++17

Optimisation



- Concentrated on speed of access to doubles
  - each double in its own branch
  - using separate branches is very common, but slower
- Optimisation options:

1. “map”: TTreeIterator initially used a `std::map` to lookup in the branch cache (returning a `BranchInfo` object)
2. “map+vector”: implements an ordered map so the search can first try the next element
   - most element accesses are in the same order for each entry
3. “vector”: keep all BranchInfo objects in a single vector to allow fast traversal
   - this is much slower for out-of-order accesses, but can use a map if first look fails
     - to be implemented
   - `std::vector` can move the vector’s data as new elements are added
     - TTreeIterator checks for a move and reissues `SetBranchAddress` if needed
       - this only happens as new values are accessed, so doesn’t significantly affect speed
4. “std::any opt”: enable optimisations implemented in `Cpp11::any`
5. “no checks”: disables some frequent checks
   - eg. that user didn’t call `SetBranchAddress` herself, so TTreeIterator doesn’t interfere
   - this doesn’t slow down access much, so the default is 4: “std::any opt”

Is this useful with PyROOT?

- PyROOT can't easily call TTreeIterator directly
  - would have to specify data type for template argument
  - PyROOT's TTree access can use type at run-time

- PyROOT's TTree access is notoriously slow
  - can we use some similar techniques to optimise
  - I tried to optimise PyROOT's TTree access along these lines many years ago
    - large speed-up, but that was for an old version of PyROOT
    - perhaps could still work
  - can save the cached access in a closure attached to Python method

Other ideas

- Would it help to use a `std::back_inserter`-style model for filling?
- Optionally, throw exception on run-time check for missing branch or wrong type?
- Various options to improve `entry["var"]` automatic type deduction
  - define simple operators, `+ – * /`
  - can we assume `double` if ambiguous?
  - automatic run-time type conversion for some pre-defined types
    - first access with that type could activate (and cache) type conversion function if type doesn’t match branch type
    - eg. `float` → `double`, `double` → `int`
  - or is it best not to use this access mode and always specify the type explicitly

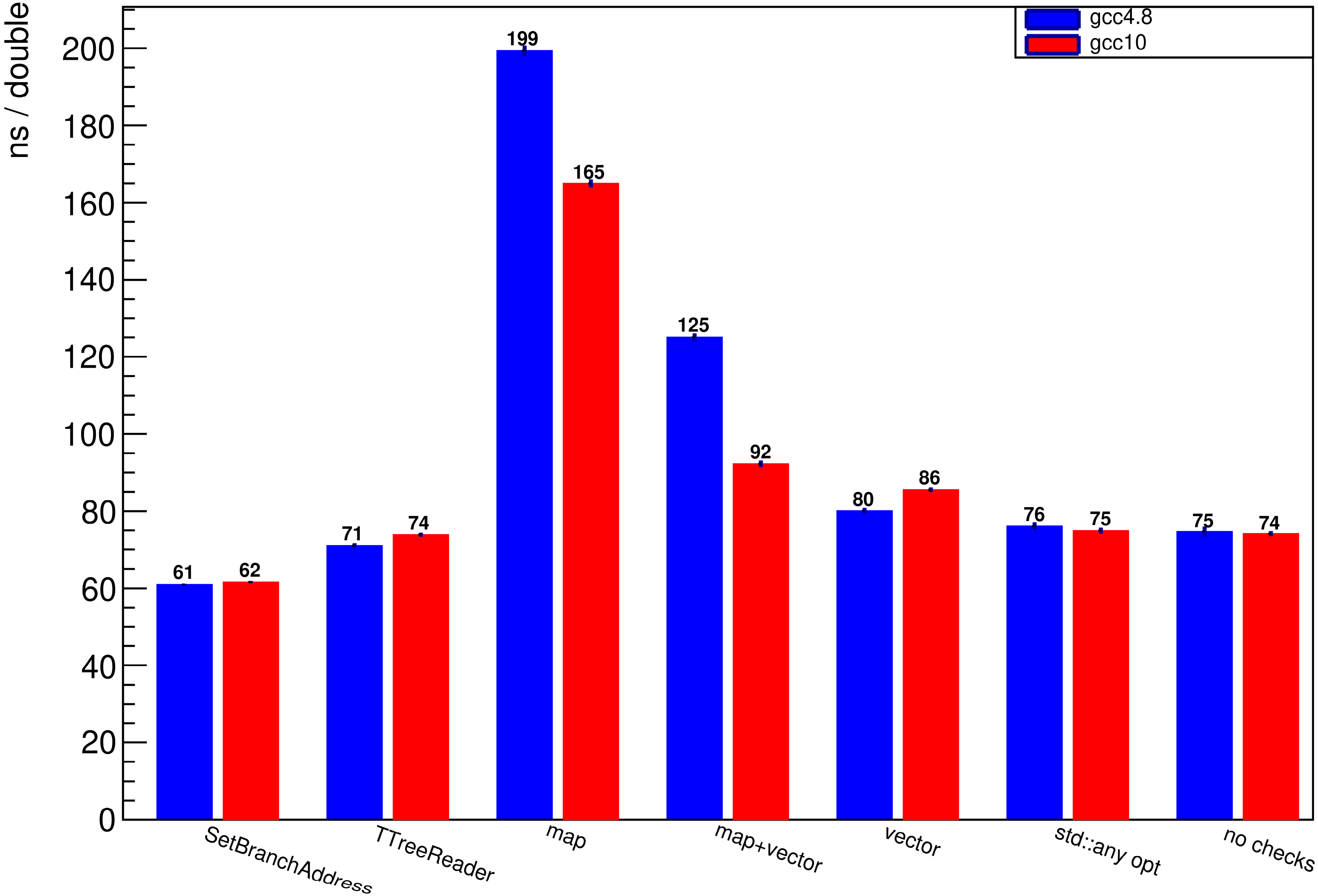
What next?

- How should I take this forward?
  - The ROOT team (quite reasonably) does not want to add another API
    - perhaps some of the ideas I discussed are still useful
    - eg. similar ideas implemented in TTreeReader or RNTuple
  - Otherwise, I can release this in a package of its own
    - header-only, so easy to install
    - code would need some cleanup
      - eg. remove `#ifdef` code for different benchmark options
    - needs documentation!

- Currently part of another project, here:
  - https://gitlab.cern.ch/will/roofittrees/-/blob/tim/RooFitTrees/RooFitTrees/TTreeIterator.h
  - with implementation here:
    https://gitlab.cern.ch/will/roofittrees/-/tree/tim/RooFitTrees/RooFitTrees/detail/

Backup

Performance measurements – C++11 vs C++17



access method or optimisation

- Each test sums 100 doubles in 500,000 entries
- Each test repeated ~10 times for ~0.5ns uncertainty on average
- Run on empty Xeon E5-2620v4 2.1GHz (8 cores, HT)
- CentOS 7.9.2009, ROOT 6.22