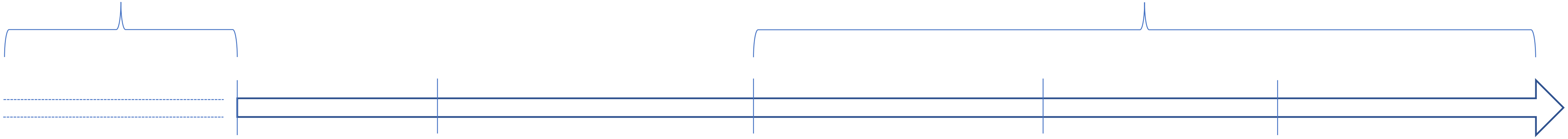
Modern C++: memory management, standard library and more

What does modern C++ mean?

Standardization

3 years cycle



1985

First version of c++

C++98

C++03

Minor revision

Nothing spectacular

C++11

Major revision

Many new language features

C++14

Minor revision

C++17

Several new language features and additions to the standard library

C++20 …

Interesting features expected

Several of them discussed in this talk

Modern c++: Utilizing features introduced since c++11

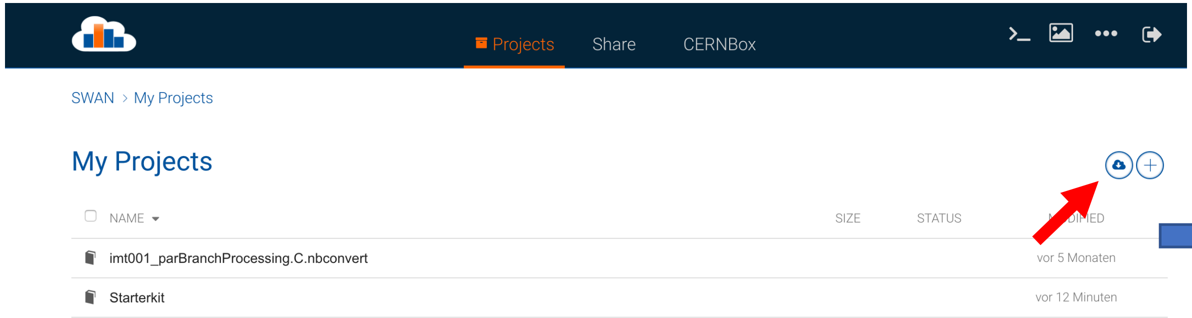
Reference guides

- https://root.cern/doc/master/index.html
- http://www.cplusplus.com/reference/

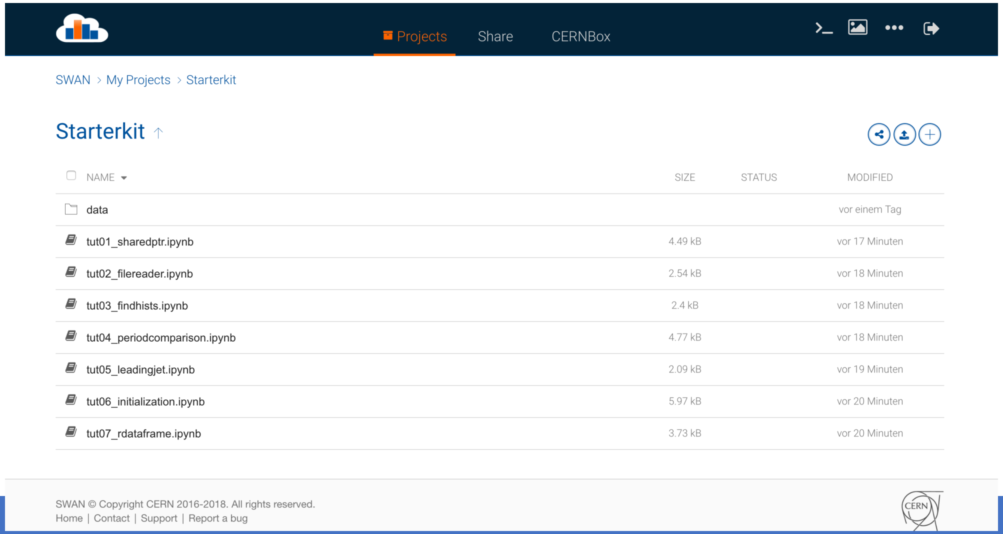
- General c++ coding guidlines with lots of examples: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

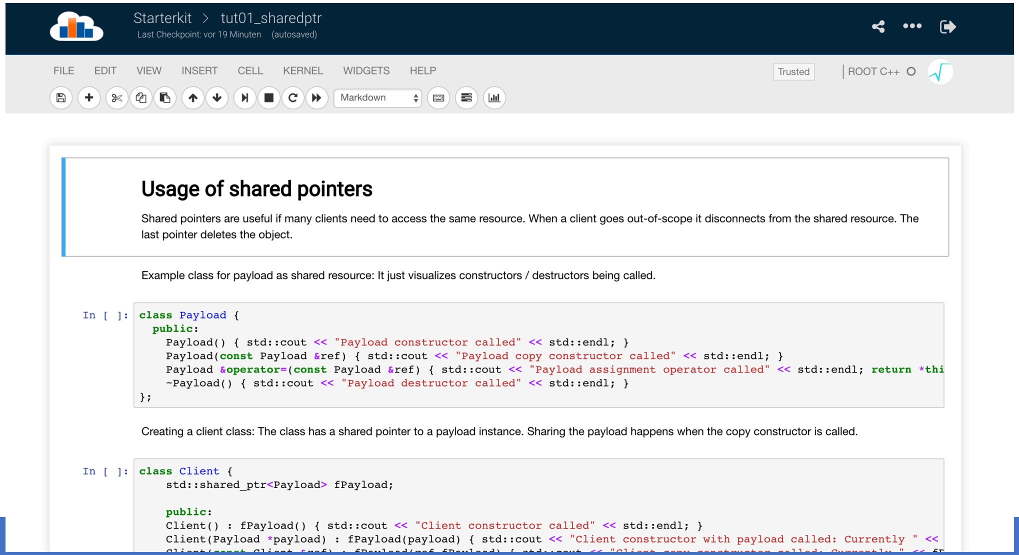
How to run the exercises

Go to swan.cern.ch and start a new session



https://github.com/mfasDa/starterkitmoderncpp



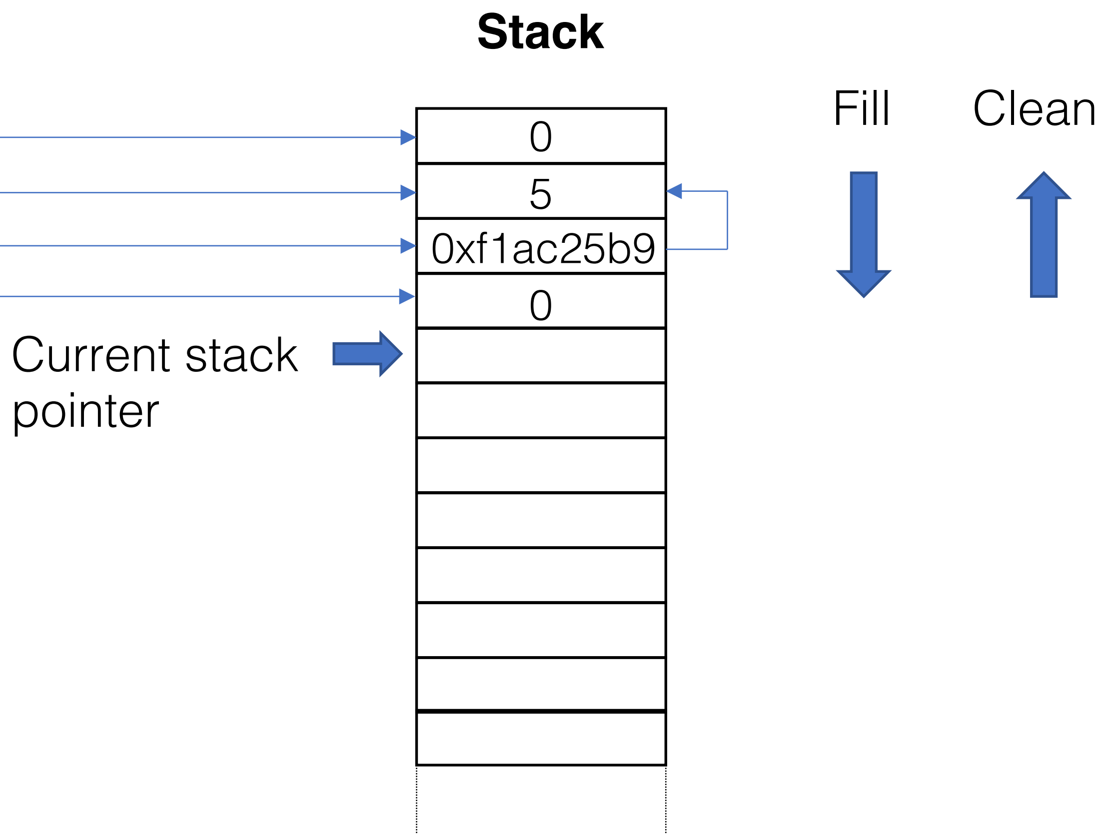


Memory management

Memory management in c++: Stack and heap

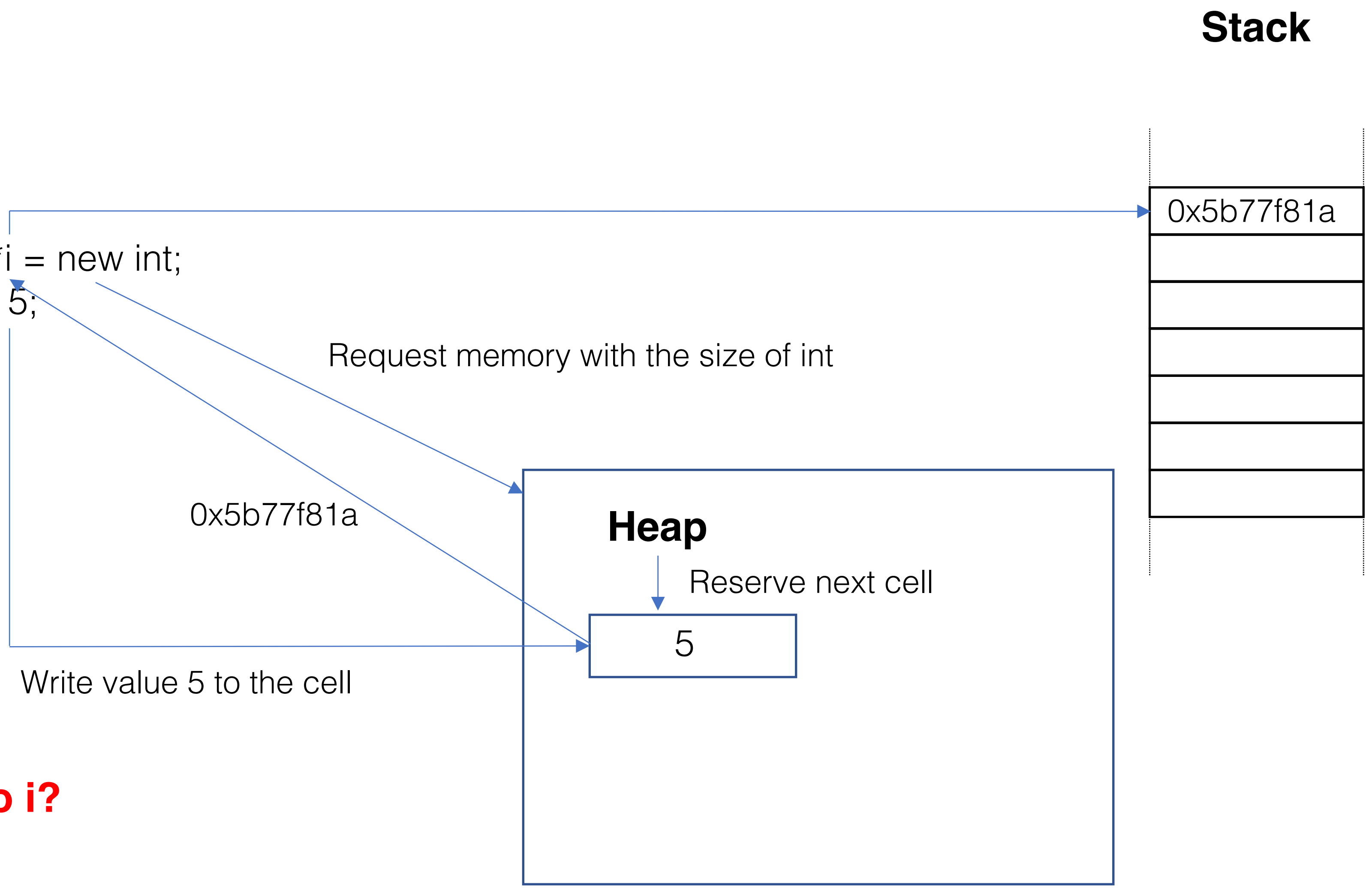

```
int main(int argc, const char ** argv) {
        int i = 5;
        int *j = &i;
        return EXIT_SUCCESS;
}
```

Closing bracket := end of scope
Start cleaning the stack



Memory management in c++: Stack and heap


```
{
        ...
        int *i = new int;
        *i = 5;
        ...
}
```



End of scope:
Start cleaning
the stack

What happens to i?

What is a memory leak?


```
...
for(int i = 0; i < fMCEvent->GetNumberOfTracks(); i++) {
        AliVParticle *part1 = fMCEvent->GetTrack(i);
        TLorentzVector *pvec1 = new TLorentzVector(part1->Px(), part1->Py(), part1->Pz(), part1->E());
        for(int j = i+1; j < fMCEvent->GetNumberOfTracks(); j++) {
                AliVParticle *part2 = fMCEvent->GetTrack(j);
                TLorentzVector *pvec2 = new TLorentzVector(part2->Px(), part2->Py(), part2->Pz(), part2->E());
                std::cout << „Distance between tracks “ << i << „ and “ << j << „: “ << pvec1->DeltaR(*pvec2) << std::endl;
        }
}
...
```

Consequences:

- Best case: Process killed by the system
- Worst case: Process starting to write to swap

Exercise: leakingProgram.C

How can we avoid memory leaks?

- Do we really need to create every object with new?
- Forget about c-arrays! Use std::vector instead
- If you create objects with new capture them with smart pointers

How does a smart pointer work

- Smart pointers are classes which behave like a pointer
- They carry the raw pointer but can also carry more!
- They live on the stack!
- The smart pointer destructor deletes the raw pointer it contains

```
template<typename t>
class unique_ptr {
public:
        unique_ptr(t *object): fObject(object) {}
        ~unique_ptr() { if(fObject) delete fObject;}
        ...
private:
        t *fObject;
};


...
for(int i = 0; i < 10; i++) {
        // Creating the smart pointer -> Constructor
        unique_ptr<HeavyPayload> obj(new HeavyPayload);
        // do something with the object
        obj->DoSomething();
} // end of scope reached for obj, Destructor called automatically
```

Exercise: Modify leakingProgram.C using std::unique_ptr<HeavyPayload> capturing new object, watch memory consumption

Difference between smart pointers

unique_ptr

- Only one pointer can point to object
- Cannot be copied
- Ownership can be passed

shared_ptr

- Multiple shared_ptr can point to same object
- Containing reference count
- New pointer (via copy): Increase reference count
- Delete: Decrease reference count
- When reference count is 0: delete object and reference counter
  - Last pointer does the delete
- Extra overhead for reference counter

Exercise: tut01_sharedptr (notebook)

When do we use which smart pointer?

- Shared pointers:
  - Object is shared by many clients (i.e. different objects)
- Unique pointers:
  - Capture pointers which are returned by a function
- Raw pointers: Only as function argumets / return values
  - Example: Handling of TFile in local functions / ROOT macros
- Be cautious with smart pointers as function arguments

Exercise: tut01_filereader (notebook)

Standard template library

Containers and iterators

Container

Data structure that can store multiple objets of the same type and provide access to it

i.e. TList, TMap, TObjArray, ...

Iterator

Pointers accessing (iterating over) all elements in the container in a predefined way

Iterating over containers

```
std:vector<std::string> data
for(std::vector<std::string>::iterator it = data.begin();
      it != data.end(); it++) {
  std::cout << *it << std::endl;
}
```

c++11

```
for(const auto &s : data) {
  std::cout << s << std::endl;
}
```

auto: Compiler determines type

stl-containers

- std::array
- std::vector
- std::map
- std:set
- std::unordered_map

examples

Searching an object inside a container

- Non-associative container (array, vector, set):

```
std::find_if(begin iterator, end iterator, condition)
```

Can be begin() or other iterator position in container

Can be end() or other iterator position in container

Function (from library, user-defined or lambda)

Lambda function:

```
auto fun = [](int x, int y) {...}
```

Capture parameter

Function arguments

Return type determined automatically

- Associative container (map,unordered_map):

Find method implemented in class, find value according to key

Always returning iterators, must be checked against end() and dereference

Task: Write a function which searches a histogram with a certain tag inside a file

Exercise: tut03_findhists

When to use which container?

Different use cases for ROOT and stl containers

ROOT containers:

- Usefull when storing objects of different type
- Some advantages with ROOTs file I/O
- Complicated for primitive datatypes
- Not type safe

stl containers:

- Supporting any type
- Type safe
- ROOT I/O a bit more complicated

Task

A crucial step in the analysis is often to compare different data samples (runs, periods, MC samples).
For this one usually draws histograms with different data on the same canvas, using different styles.
In order to reduce code duplication a style class can handle common steps, and all users have to do is to create a list of styles. Create a style definition in the tutorial, either using std::map or ROOT‘s TMap

Exercise: tut04_periodcomparison

Function object

c++11: std::function

std::function<return type (function argument types)>

```
auto fun = [](int x, int y) { return x + y; };
```

fun: std::function<int (int, int)>

Use functions as function arguments
Store functions in lists

Function objects can be return types of other functions

Outer function

Inner function

```
auto style = [](Color_t col, Style mrk) {
        return [col, mrk] (TH1 *hist) {
                hist->SetMarkerColor(col);
                hist->SetMarkerStyle(mrk);
                hist->SetLineColor(col)
        }
};
```

Capture parameters from outer function

Closure

C++14: Generic lambdas

```
auto mylambda = [] (auto par1, auto par2) {
        return par1 + par2;
};
```

Determined by the compiler

Can be all types supporting operator+

Exercise: tut05_genericstyles

Sorting containers

std::sort(begin iterator, end iterator, comparator)

Can be begin() or other iterator position in container

Can be end() or other iterator position in container

Possible Comparators:

- std::less<type> - sorts in increasing order
- std::greater<type> - sorts in increasing order
- Any function comparing two instances of the type storing in the container (including lambda functions)
- Object implementing operator()

Task:

An event contains multiple jet candidates. Write a program that finds the two leading jets in an event. Use a std::vector to store the jet candidates and a lambda function to compare the two jets

Exercise: tut06_leadingjet

Operator overloading

Operators can be overloaded similiar to regular functions

```
class Track {
        Double_t                fPt;
        ...
public:
        ...
        Bool_t operator==(const Track &other) const { return fPt == other.fPt; }
        Bool_t operator<(const Track &other) const { return fPt < other.fPt; }
        ...
};

Track track1(5.), track2(10.);
if(track1 < track2) {                   // What happens here?
        ...
}
```

Which operators can you overload?
Are they always class members?

Exercise: Solve tut06_leadingjet using operator overloading

Object initialization

- Classes (with constructors): () and {} with arguments matching to certain constructors
  - Fixed amount of arguments
  - Variable amount of SAME TYPE arguments: initializer lists
- POD objects (only simple structs)

Exercise: tut07_initialization

Multiple return values

Old way: Return by reference

Result of the function

Return value: status for error handling

```
Bool_t GetNumberOfTPCClusters(const AliVTrack *const trk, Int_t &nclusters) {
        ...
}
```

New way: Multiple return values of different type -> std::tuple

```
/// @brief Function getting the numer of clusters in the TPC from a track
/// @return tuple <int, bool> with
///     - Number of clusters
///     - Error status
std:tuple<int, bool> GetNumberOfTPCClusters(const AliVTrack *const trk) {
        if(!trk) return std::make_tuple(0, false);
        if(!(trk->GetStatus() & AliVTrack::kTPCrefit) return std::make_tuple(0, false);
        return std::make_tuple(trk->GetTPCncls(), true);        // Create the tuple
}
...
auto clusterres = GetNumberOfTPCClusters(trk);
if(!std::get<1>(clusterres)) continue;                          // access to tuple element
hClusters->Fill(std::get<0>(clusterres);
```

Attention: Introduced in c++11, special treatment in headers for ROOT5 compatibility

C++11 code in ALICE headers

AliRoot/AliPhysics still required to be compatible with ROOT5

C++11 not supported by CINT/ROOTCINT, need to be excluded from (ROOT)CINT

```
#if !(defined(__CINT__) || defined(__MAKECINT__))
// your C++11 code goes here
#endif
```

No implications for ROOT6

So much more one could talk about ...

- Multi-threading
- constexpr
- default/delete for constructors/destructors/operators
- final/override for virtual functions
- enum classes

Two highlights from c++17

Structured binding

C++11

```
std::map<int, int> mymap;
for(auto en : mymap) {
        hist->Fill(en.second);
}
```

With c++17 this reduces to

```
std::map<int, int> mymap;
for(auto [k,v] : mympa) hist->Fill(v);
```

Works for:

- maps (pairs)
- tuples
- Structs (PoD objects)

String view

- std::string: owns data (char array), additional functionality
- std::string_view: Only wraps around a c-string (char array), but provides same functionality as a string

Can improve code speed for non-owning strings (i.e. as function arguments)

Modern ways to process ROOT trees

Evolution of tree processing in ROOT

ROOT5

```
TTree *t = ...;
Double_t px, py px;
t->SetBranchAddress(„px“, &px);
t->SetBranchAddress(„py“, &py);
t->SetBranchAddress(„pz“, &pz);
for(int i = 0; i < t->GetEntries(); i++){
   t->GetEntry(i);
   hpx->Fill(px);
   ...
}
```

Since ROOT6

```
TTree *t = ...;
TTreeReader reader(t);
TTreeReaderValue<double> px(reader, „px“),
                         py(reader, „py“),
                         px(reader, “pz“);
for(auto en : reader) {
   hpx->Fill(*px);
   ...
}
```

Evolution of tree processing in ROOT

ROOT5

```
TTree *t = ...;
Double_t px, py px;
t->SetBranchAddress(„px“, &px);
t->SetBranchAddress(„py“, &py);
t->SetBranchAddress(„pz“, &pz);
for(int i = 0; i < t->GetEntries(); i++){
    t->GetEntry(i);
    hpx->Fill(px);
    ...
}
```

RDataFrame

ROOT6

```
e *t = ...;
reeReader reader(t);
TTreeReaderValue<double> px(reader, „px“),
                         py(reader, „py“),
                         px(reader, “pz“);
for(auto en : reader) {
    hpx->Fill(*px);
    ...
}
```

RDataFrame

Declarative programming: express program flow as chain of high-level operations

```
ROOT::RDataFrame df(„testtree“, „testfile.root“);
auto hist = df.Histo1D({„hPx“, „hPx“, 100, -50., 50.}, “px“);
hist->Draw();
```

Name of the branch / coloum

Only declare operation to be performed

Histogram model

Execute operation, draw histogram

What is the type of hist?

https://root.cern/doc/master/classROOT_1_1RDataFrame.html

Filter / Define operations

Define

New branch/column

Expression defining branch

Branches needed in expression

```
auto framewithpt = df.Define("pt", [](double px, double py) { return TMath::Sqrt(px*px + py*py)); }, {„px", „py"});
auto hpt = framewithpt.Histo1D({"hpt", „hpt", 100, 0., 100.}, „pt");
hpt->Draw();
```

Simple expression or c++11 lambda function

Filter

```
auto highpt = framewithpt.Filter(„pt > 10");
highpt.Histo1D(...);
```

Exercise: tut08_rdataframe

Going parallel

For large datasets one want to utilize all cores on a machine ➡ Multi-threading

Explicit multi-threading complicated (synchronization, thread safety ...) ➡ Multi-processing

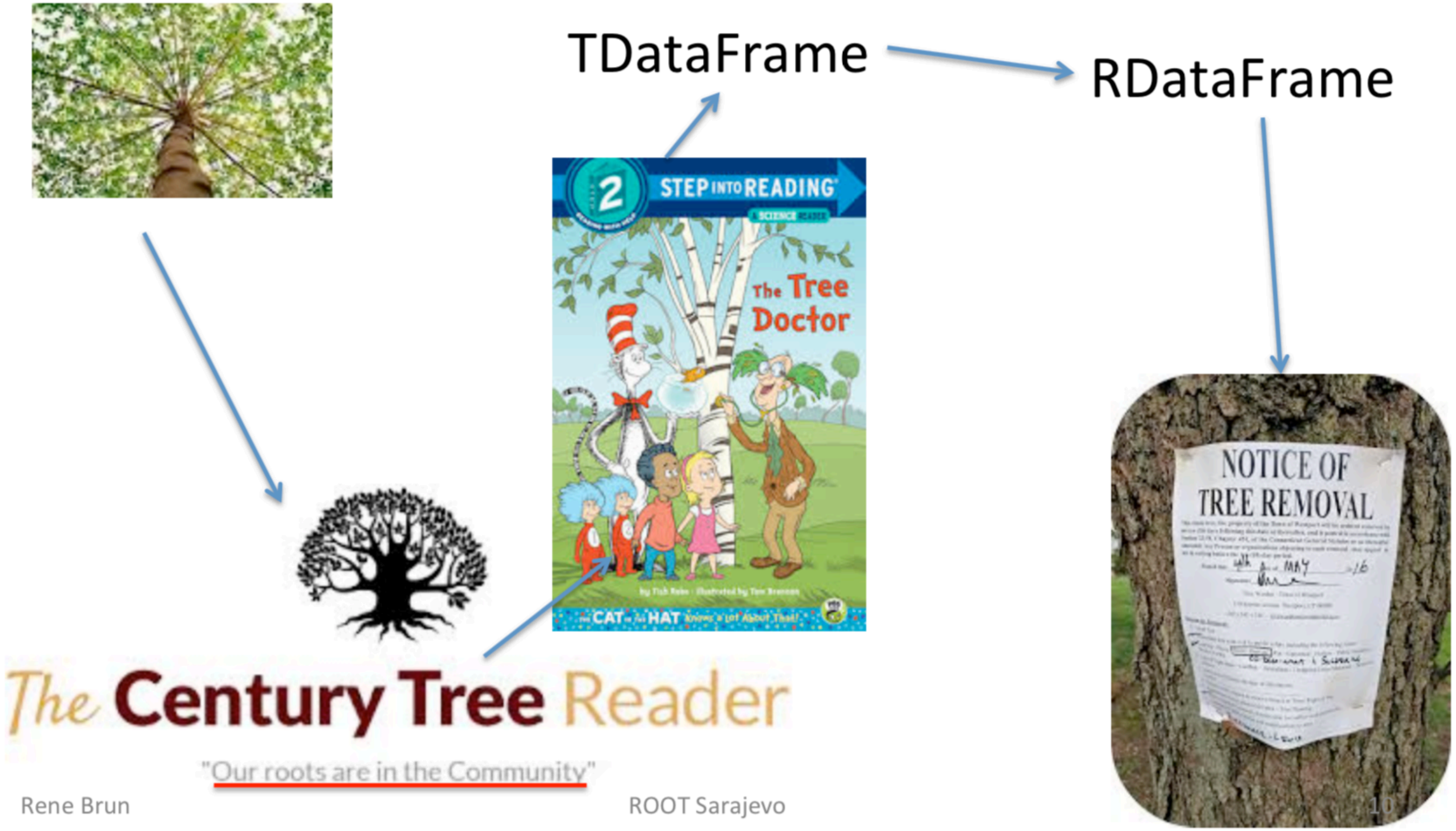
TProcessExecutor (https://root.cern/doc/master/classROOT_1_1TProcessExecutor.html)

Still needs dedicated code. Can this be automatized?

ROOT::EnableImplicitMT(numberofworkers);

→ Code generated by RDataFrame running multi-threaded with n-cores

Warning!!!!
The young developer syndrome



R. Brun, ROOT Users Workshop 2018
https://indico.cern.ch/event/697389/contributions/3102464/attachments/1712676/2761669/Brun_Sarajevo.pdf

Take away

- Use dynamic memory allocation only when needed
- Use smart pointers to manage the lifetime of objects allocated dynamically
- Double_t *... = new Double_t[]; ⟹ std::vector<Double_t>
- The compiler is your friend. Let him help you spotting bugs!
- Consider using the standard library – it provides helpful tools to many common tasks
- RDataFrame simplifies handling with ROOT trees and allows exploiting multicore systems without dedicated code from the user