Modern C++: memory management, standard library and more
What does modern C++ mean?

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 guidelines 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
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

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

End of scope: Start cleaning the stack

What happens to i?
What is a memory leak?

For example:

```cpp
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
**Differences 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 arguments / 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 objects 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**

```cpp
std::vector<std::string> data

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

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

**c++11**

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

- 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 trigger detector selects an event if at least one particle deposits a minimum energy in the detector.

Exercise: tut04_periodcomparison
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: tut05_leadingjet
Operators can be overloaded similar to regular functions

```cpp
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 tut05_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: tut06_initialization
Multiple return values

Old way: Return by reference

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

Return value: status for error handling

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

```cpp
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
}
```

Attention: Introduced in c++11, special treatment in headers for ROOT5 compatibility
C++11 not supported by CINT/ROOTCINT, need to be excluded from (ROOT)CINT

```cpp
#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
- std::string_view (C++17)
Modern ways to process ROOT trees
Evolution of tree processing in ROOT

ROOT5

```c++
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

```c++
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**

```cpp
TTree *t = ...;
Double_t px, py, pz;
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);
    ... 
}
```

**ROOT6**

```cpp
RDataFrame rdf = ...
TTreeReader reader(t);
TTreeReaderValue<
    Double_t>
    px(reader, "px"),
    py(reader, "py"),
    pz(reader, "pz");
for(auto en : reader) {
    hpx->Fill(px);
    ... 
}
```
Declarative programming: express program flow as chain of high-level operations

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

What is the type of hist?

Name of the branch / column

Only declare operation to be performed

Histogram model

 Execute operation, draw histogram

https://root.cern/doc/master/classROOT_1_1RDataFrame.html
**Filter / Define operations**

**Define**

- New branch/column
- Expression defining branch
- Branches needed in expression

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

**Filter**

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

**Exercise:** tut07_rdataframe
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

TDataFrame

RDataFrame

The Century Tree Reader
"Our roots are in the Community"

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[]; \Rightarrow \text{std}::\text{vector}\langle\text{Double}_t\rangle
• 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