NAIA

Release 0.1

Valerio Formato

May 09, 2022

NAIA

This is the official documentation page for the **NAIA** project. The project focuses on providing a common data format for AMS analysis that can be shared by multiple groups.

CHAPTER

ONE

DOCS:

Take a look at:

- *Getting started*
- *The NAIA data model*
- *The Event class*
- *Skimming*
- *Examples*

1.1 Getting started

1.1.1 Requirements

To use NAIA you'll need:

- A C++ compiler with full c++14 support (tested with gcc >= 9.3.0)
- CMake version >= 3.13
- A ROOT installation compiled with c++14 support (tested with ROOT >= 6.22/08)

If you have access to cvmfs then you can find all the requirements in:

```
/cvmfs/ams.cern.ch/Offline/amsitaly/public/install/x86_64-centos7-gcc9.3/naia
```

and a `setenv` script is already provided with each NAIA version, e.g. for CentOS7:

```
/cvmfs/ams.cern.ch/Offline/amsitaly/public/install/x86_64-centos7-gcc9.3/naia/v0.1.0/
↪setenvs/setenv_gcc6.22_cc7.sh
```

Note: For the ntuple production some additional requirements are needed:

- A gbatch installation compiled with
  - `export NOCXXSTD=1` (gbatch hardcodes `-std=c++11` in the Makefile... This variable prevents that)
  - `export GLIBCXX_USE_CXX11=1` (gbatch hardcodes the old gcc ABI in the Makefile... Most likely someone didn't know what he was doing)
  - Run `CPPFLAGS="-std=c++14" make lib` to build the gbatch library (if you don't want to hack the Makefile and change the C++ standard manually)

1.1.2 Building and installing

Follow this simple procedure:

- Clone this repository
  - `git clone https://:@gitlab.cern.ch:8443/ams-italy/naia.git -b v0.1.1` (Kerberos)
  - `git clone ssh://git@gitlab.cern.ch:7999/ams-italy/naia.git -b v0.1.1` (SSH)
  - `git clone https://gitlab.cern.ch/ams-italy/naia.git -b v0.1.1` (HTTPS)
- Create a build and install directory
  - e.g: `mkdir naia.build naia.install`
- Build the project
  - `cd naia.build`
  - `cmake ../naia -DCMAKE_INSTALL_PREFIX=${your-install-path-here}` (for ntuple production add the `-DPRODUCTION_CODE=ON` arg)
  - `make all install`

1.1.3 Using the project

To use the NAIA ntuples your project needs:

- the headers in `naia.install/include`
- the `naia.install/lib/libNAIAUtility.so` library
- the `naia.install/lib/libNAIAContainers.so` library
- the `naia.install/lib/libNAIAChain.so` library

The recommended way of using NAIA in your project is to use CMake and let it do all the heavy lifting for you. NAIA targets are set up so that required includes and libraries are automatically passed to your targets. In your `CMakeLists.txt` you just need:

```
find_package(NAIA REQUIRED)

set(SOURCES MyProgram.cpp)

add_executable(MyProgram ${SOURCES})
target_link_libraries(MyProgram NAIA::NAIAChain)
```

and you should be good to go.

Alternatively you can set up your own makefile and do all the work manually. For this see the examples provided in the NAIA repository.

Note: If you are using a pre-installed NAIA distribution (e.g. from cvmfs) you might have to export the `ROOT_INCLUDE_PATH` variable to include the path of the NAIA headers.

```
export ROOT_INCLUDE_PATH=${path-to-the-NAIA-install}/include:$ROOT_INCLUDE_PATH
```

This is due to ROOT needing to parse the headers at runtime. (see for example)

1.1.4 Included facilities

These two libraries are automatically build with the project and included in the installation so that they could be used out-of-the-box

fmt

See https://github.com/fmtlib/fmt

This is a library for text formatting that implements the formatting specification introduced in the C++20 standard, the syntax is similar to the python format() function. It's a header-only library that is always lighter and faster than using iostream (example).

Note: It is incredibly useful and flexible once you get used to the syntax (and it's way better than littering your code with thousands of <<)

---

spdlog

https://github.com/gabime/spdlog

This is a header-only library for asynchronous logging build on top of `fmt` which allows to quickly log messages from a program with different levels of depth, customization and filtering.

Note: It can be useful saving you from several `if(DEBUG) std::cout << "debug statement" << std::endl;` :)

Note: For any question or in case you need help write to valerio.formato@cern.ch

---

1.2 The NAIA data model

The NAIA data model is vaguely inspired by gbatch. The first thing needed to access data is to create a `NAIAChain` object

```
// ...
#include "Chain/NAIAChain.h"

int main(int argc, char const *argv[]) {
  NAIA::NAIAChain chain;
  chain.Add("somefile.root");
  chain.SetupBranches();
}
```

the `chain.SetupBranches()` is mandatory (with some work it could be made automatic with the instantiation of a chain, but this might come in a future release) and takes care of setting up the whole "read-on-demand" mechanism.

1.2.1 Looping

The chain contains all the events in the added runs, looping over events is particularly easy:

```
for(Event& event : chain){
  // your analysis here :)
}
```

If you're uncomfortable with range-based for loops you can still do it the old fashioned way

```
unsigned long long nEntries = chain.GetEntries());

for (unsigned long long iEv = 0; iEv < nEntries; iEv++) {
  Event &event = chain.GetEvent(iEv);

  // your analysis here :)
}
```

NAIA root-files contain two more `TTree` with additional data for the analysis.

The `RTIInfo` tree

The data about the ISS position, its orientation, and physical quantities connected to them, as well as some time-averaged data about the run itself are usually retrieved in AMS analysis from the RTI (Real Time Information) database. This database stores data with a time granularity of one second, and it can be accessed using the gbatch library.

Since we try to get rid of any dependency on gbatch during the analysis the entire RTI database is converted to a `TTree` that is stored alongside the main event `TTree` in the NAIA root-files. This tree has only one branch, which contains objects of the `RTIInfo` class, one for each second of the current run.

When looping over the events you can get the `RTIInfo` object for the current event by calling

```
NAIA::RTIInfo &rti_info = chain.GetEventRTIInfo();
```

In some cases you might not want to loop over all the events, but still perform analysis on the RTI data standalone. In such cases you can directly retrieve the RTI tree from the NAIA file and loop over each second.

```
TChain* rti_chain = chain.GetRTITree();
NAIA::RTIInfo* rti_info = new RTIInfo();
rti_chain->SetBranchAddress("RTIInfo", &rti_info);

for(unsigned long long isec=0; isec < rti_chain->GetEntries(); ++isec){
  rti_chain->GetEntry(isec);

  // your analysis here :)
}
```

The `FileInfo` tree

In a similar fashion we also store some useful information about the original AMSRoot file that from which the current NAIA file was derived. This information is stored in the FileInfo `TTree`, which usually has only a single entry for each NAIA root-file. Having this data in a `TTree` allows us to chain multiple NAIA root-files and still be able to retrieve the FileInfo data for the current run we're processing.

This tree has one branch, which contains objects of the `FileInfo` class and, if the NAIA root-file is a Montecarlo file, an additional branch containing objects of the `MCFileInfo` class.

When looping over the events you can get the `FileInfo` object for the current event by calling

```
NAIA::FileInfo &file_info = chain.GetEventFileInfo();
```

Also in this case you can directly retrieve the FileInfo tree from the NAIA file and loop over each entry.

```
TChain* file_chain = chain.GetFileInfoTree();
NAIA::FileInfo* file_info = new NAIA::FileInfo();
NAIA::MCFileInfo* mcfile_info = new NAIA::MCFileInfo();

file_chain->SetBranchAddress("FileInfo", &file_info);
if(chain.IsMC()){
  file_chain->SetBranchAddress("MCFileInfo", &mcfile_info);
}

for(unsigned long long i=0; i < file_chain->GetEntries(); ++i){
  file_chain->GetEntry(i);

  // do stuff with file_info

  if(chain.IsMC()){
    // do stuff with mcfile_info
  }
}
```

1.2.2 Containers

The main structure for holding data in the NAIA data model is the *Container*. Each container is associated to a single branch in the main `TTree` and allows for reading the corresponding branch data only when first accessed.

This means that if you never use a particular container in your analysis, you'll never read the corresponding data from file

Note: i.e.: `TBranch::GetEntry` will never be called unless actually needed

Warning: In order for this to work in NAIA we overload the `->` operator to hide this "read-on-demand" behavior. It is required that you always use `->` to access the data members and methods of a container.

Example:

```
// Get the inner tracker charge from the "trTrackBase" container
auto innerCharge = event.trTrackBase->Charge[NAIA::TrTrack::ChargeRecoType::YJ];
//                                        ^^
//                                this is very important :)
```

Variable types and structure

Most variables in AMS analysis are computed for several different variants, which usually refer to different possible reconstructions of the same quantity. To mantain the data format as light as possible, and not write to disk non-existing data, variables in NAIA are often implemented as associative containers (e.g: `std::map`).

If that is the case, then there is always a `enum` describing all the available variants for a given variable.

If you want to make sure that a given variant exists you can use the `ContainsKeys` function. This function takes a container and one or more keys and will check recursively that those keys exist in the container structure.

```
if (NAIA::ContainsKeys(event.tofBase->Charge, NAIA::Tof::ChargeType::Upper))
  tof_charge = event.tofBase->Charge[NAIA::Tof::ChargeType::Upper];
```

because it is not guaranteed that, for example, a particular reconstruction succeeded, or that there is a hit on a given layer.

Note: The `KeyExists` function is completely replaced by `ContainsKeys`. It is still available for backward-compatibility but it is now deprecated and will be removed in a future release. A warning message will be printed (at most 10 times), advising to switch to `ContainsKeys`.

As an example, what before would have been achieved with

```
if (KeyExists(layer, LayerCharge) && KeyExists(NAIA::Track::ChargeRecoType::YJ,␣
↪LayerCharge.at(layer)) &&
  KeyExists(TrTrack::Side::X, LayerCharge.at(layer).at(NAIA::Track::ChargeRecoType::YJ)))
```

is now done by

```
if (ContainsKeys(LayerCharge, layer, NAIA::Track::ChargeRecoType::YJ, TrTrack::Side::X))
```

Note: Not all variables are stored in associative containers, when we know that all possible variants of a variable will be present we use a `std::vector` instead.

In NAIA there are several variable archetype defined, so that it is clear which `enum` to use and what kind of variable variant is available. The archetypes in the NAIA data model are:

- `LayerVariable`: one number for each layer (applies to Tracker, Tof, TRD, ...).
  - Uses the layer number `(0, ..., N-1)` for access
  - ```
    template<class T>
    using LayerVariable = std::map< unsigned int, T >
    ```
  - Example:

    ```
    unsigned int layer = 4; // layer 5
    if (NAIA::ContainsKeys(event.trTrackPlus->TrackFeetDistance, layer))
      track_distance_to_feet_l5 = event.trTrackPlus->TrackFeetDistance[layer];
    ```
- `EcalEnergyVariable`: one number for each energy reconstruction type.
    ```

  - Uses the `Ecal::EnergyRecoType` enum for access
  - ```
    template<class T>
    using EcalEnergyVariable = std::vector< T >
    ```
  - Example:

    ```
    if (NAIA::ContainsKeys(event.ecalBase->Energy, NAIA::Ecal::EnergyType::EnergyD))
      ecal_energy_D = event.ecalBase->Energy[NAIA::Ecal::EnergyType::EnergyD];
    ```

- `EcalLikelihoodVariable`: one number for each likelihood type.
  - Uses the `Ecal::LikelihoodType` enum for access
  - ```
    template<class T>
    using EcalLikelihoodVariable = std::vector< T >
    ```
  - Example:

    ```
    if (NAIA::ContainsKeys(event.ecalPlus->Likelihood,␣
    ↪NAIA::Ecal::Likelihood::Integral))
      ecal_likelihood = event.ecalPlus->
    ↪Likelihood[NAIA::Ecal::Likelihood::Integral];
    ```

- `EcalBDTVariable`: one number for each BDT type.
  - Uses the `Ecal::BDTType` enum for access
  - ```
    template<class T>
    using EcalBDTVariable = std::vector< T >
    ```
  - Example:

    ```
    if (NAIA::ContainsKeys(event.ecalBase->BDT, NAIA::Ecal::BDTType::v7std))
      bdt = event.ecalBase->BDT[NAIA::Ecal::BDTType::v7std];
    ```

- `RichBetaVariable`: one number for each RICH beta reconstruction type.
  - Uses the `Rich::BetaType` enum for access
  - ```
    template<class T>
    using RichBetaVariable = std::map< Rich::BetaType, T >
    ```
  - Example:

    ```
    if (NAIA::ContainsKeys(event.richBase->GetBeta(), NAIA::Rich::BetaType::CIEMAT))
      rich_beta = event.richBase->GetBeta()[NAIA::Rich::BetaType::CIEMAT];
    ```

- `TofChargeVariable`: one number for each kind of Tof charge.
  - Uses the `Tof::ChargeType` enum for access
  - ```
    template<class T>
    using TofChargeVariable = std::map< Tof::ChargeType, T >
    ```
  - Example:
    ```

```
if (NAIA::ContainsKeys(event.tofBase->Charge, NAIA::Tof::ChargeType::Upper))
  tof_charge = event.tofBase->Charge[NAIA::Tof::ChargeType::Upper];
```

- `TofBetaVariable`: one number for each Tof beta reconstruction type.
  - Uses the `Tof::BetaType` enum for access
  - ```
    template<class T>
    using TofBetaVariable = std::map< Tof::BetaType, T >
    ```
  - Example:

    ```
    if (NAIA::ContainsKeys(event.tofBase->Beta, NAIA::Tof::BetaType::BetaH))
      tof_beta = event.tofBase->Beta[NAIA::Tof::BetaType::BetaH];
    ```

- `TofClusterTypeVariable`: one number for each Tof cluster type.
  - Uses the `Tof::BetaClusterType` enum for access
  - ```
    template<class T>
    using TofClusterTypeVariable = std::map< Tof::BetaClusterType, T >
    ```
  - Example:

    ```
    unsigned int layer = 0;
    if (NAIA::ContainsKeys(event.tofPlus->Nclusters, layer,␣
    ↪NAIA::Tof::BetaClusterType::OnTime))
      ontime_clusters = event.tofPlus->
    ↪NClusters[layer][NAIA::Tof::BetaClusterType::OnTime];
    ```

- `TrdChargeVariable`: one number for each TRD charge reconstruction type.
  - Uses the `TrdK::ChargeType` enum for access
  - ```
    template<class T>
    using TrdChargeVariable = std::vector< T >
    ```
  - Example:

    ```
    if (NAIA::ContainsKeys(event.trdKBase->Charge, NAIA::TrdK::ChargeType::Total))
      trd_charge = event.trdKBase->Charge[NAIA::TrdK::ChargeType::Total];
    ```

- `TrdLikelihoodVariable`: one number for each TRD likelihood type.
  - Uses the `TrdK::LikelihoodType` enum for access
  - ```
    template<class T>
    using TrdLikelihoodVariable = std::vector< T >
    ```
  - Example:

    ```
    if (NAIA::ContainsKeys(event.trdKBase->Likelihood,␣
    ↪NAIA::TrdK::LikelihoodType::Electron))
      trd_like_e = event.trdKBase->Likelihood[NAIA::TrdK::LikelihoodType::Electron];
    ```

- `TrdLikelihoodRVariable`: one number for each TRD likelihood ratio type.
  - Uses the `TrdK::LikelihoodRType` enum for access

  - ```
    template<class T>
    using TrdLikelihoodRVariable = std::vector< T >
    ```

  - Example:

    ```
    if (NAIA::ContainsKeys(event.trdKBase->LikelihoodRatio,␣
    ↪NAIA::TrdK::LikelihoodRType::ep))
      trd_likeratio_ep = event.trdKBase->
    ↪LikelihoodRatio[NAIA::TrdK::LikelihoodRType::ep];
    ```

- `TrdOnTrackVariable`: one number for on-track / off-track TRD hits.
  - Uses the `TrdK::QualType` enum for access
  - ```
    template<class T>
    using TrdOnTrackVariable = std::vector< T >
    ```

  - Example:

    ```
    if (NAIA::ContainsKeys(event.trdKBase->NHits, NAIA::TrdK::QualType::OffTrack))
      offtrack_hits = event.trdKBase->NHits[NAIA::TrdK::QualType::OffTrack];
    ```

- `TrackChargeVariable`: one number for each Tracker charge reconstruction type.
  - Uses the `TrTrack::ChargeRecoType` enum for access
  - ```
    template<class T>
    using TrackChargeVariable = std::map< TrTrack::ChargeRecoType, T >
    ```

  - Example:

    ```
    if (NAIA::ContainsKeys(event.trTrackBase->InnerCharge,␣
    ↪NAIA::TrTrack::ChargeRecoType::YJ))
      trtrack_charge_inner = event.trtrackBase->
    ↪InnerCharge[NAIA::TrTrack::ChargeRecoType::YJ];
    ```

- `TrackFitVariable`: one number for each track fitting type, and for each track span type.
  - Uses the `TrTrack::Fit` and `TrTrack::Span` enums for access
  - ```
    template<class T>
    using TrackFitVariable = std::map< TrTrack::Fit, std::map< TrTrack::Span, T >>
    ```

  - ---

    Note: For this kind of variable you can use `TrTrackBase::FitIDEsists(TrTrack::Fit fit, TrTrack::Span span)` to check if a given fit+span combination exists

    ---

  - Example:

    ```
    if (event.trTrackBase->FitIDExists(NAIA::TrTrack::Fit::Kalman,␣
    ↪NAIA::TrTrack::Span::InnerL1))
      trtrack_rigidity_innerL1 = event.trtrackBase->
    ↪RigidityCorr[NAIA::TrTrack::Fit::Kalman][NAIA::TrTrack::Span::InnerL1];
    ```

- `TrackFitOnlyVariable`: one number for each Track fit type.

  - Uses the `TrTrack::Fit` enum for access
  - ```
    template<class T>
    using TrackFitOnlyVariable = std::map< TrTrack::Fit, T >
    ```
  - Example:

```
unsigned int layer = 1; // exclude layer 2
if (NAIA::ContainsKeys(event.trTrackPlus->PartialRigidity, layer,␣
↪NAIA::TrTrack::Fit::Choutko))
  ontime_clusters = event.trTrackPlus->
↪PartialRigidity[layer][NAIA::TrTrack::Fit::Choutko];
```

- `TrackSideVariable`: one number for each Tracker side.
  - Uses the `TrTrack::Side` enum for access
  - ```
    template<class T>
    using TrackSideVariable = std::map< TrTrack::Side, T >
    ```
  - Example:

```
if (NAIA::ContainsKeys(event.trTrackPlus->TrTrackHitPos, layer,␣
↪NAIA::TrTrack::Side::X))
  ontime_clusters = event.trTrackPlus->
↪TrTrackHitPos[layer][NAIA::TrTrack::Side::X];
```

- `TrackFitPosVariable`: one number for each fixed z-position in the Tracker.
  - Uses the `TrTrack::FitPositionHeight` enum for access
  - ```
    template<class T>
    using TrackFitPosVariable = std::map< TrTrack::FitPositionHeight, T >
    ```
  - Example:

```
auto fit = NAIA::TrTrack::Fit::Kalman;
auto span = NAIA::TrTrack::Span::InnerL1;

if (NAIA::ContainsKeys(event.trtrackPlus->TrTrackFitPos,␣
↪NAIA::FitPositionHeight::TofLayer0)){
  if (event.trTrackBase->FitIDExists(fit, span)){
    trtrack_position_at_upper_tof_x = event.trtrackPlus->
↪TrTrackFitPos[NAIA::FitPositionHeight::TofLayer0][fit][span][NAIA::TrTrack::Side::X];
↪
  }
}
```

- `TrackDistanceVariable`: one number for each distance-from-the-track type.
  - Uses the `TrTrack::DistanceFromTrack` enum for access
  - ```
    template<class T>
    using TrackDistanceVariable = std::map< TrTrack::DistanceFromTrack, T >
    ```
  - Example:

```
unsigned int layer = 1; // layer 2
if (NAIA::ContainsKeys(event.trTrackPlus->NClusters, layer,␣
↪NAIA::TrTrack::DistanceFromTrack::Onecm, TrTrack::Side::X))
  track_clusters_within_onecm_x = event.trTrackPlus->
↪NClusters[layer][NAIA::TrTrack::DistanceFromTrack::Onecm][TrTrack::Side::X];
```

- `HitChargeVariable`: same as `TrackChargeVariable`

Please refer to the doxygen documentation for all the details.

1.3 The `Event` class

The `Event` class is nothing more than a collection of containers

Table 1: Event class layout

Container type	Name	Description
Header	header	Contains simple information like run number, run tag, event number and UTC time.
EventSummary	evSummary	Contains some aggregated variables to roughly describe the event.
DAQ	daq	Contains variables describing the status of the AMS DAQ system for the event.
TofBase	tofBase	Contains basic Tof variables that are accessed most frequently
TofPlus	tofPlus	Contains additional Tof variables that are accessed less frequently
TofBaseStandalone	tofBaseSt	Contains basic Tof variables that are accessed most frequently (no-tracker reconstruction)
TofPlusStandalone	tofPlusSt	Contains additional Tof variables that are accessed less frequently (no-tracker reconstruction)
EcalBase	ecalBase	Contains basic Ecal variables that are accessed most frequently
EcalPlus	ecalPlus	Contains additional Ecal variables that are accessed less frequently
TrTrackBase	trTrackBase	Contains basic Track variables that are accessed most frequently
TrTrackPlus	trTrackPlus	Contains additional Track variables that are accessed less frequently
TrdKBase	trdKBase	Contains basic TRD variables that are accessed most frequently
TrdKBaseStandalone	trdKBaseSt	Contains basic TRD variables that are accessed most frequently (no-tracker reconstruction)
RichBase	richBase	Contains basic RICH variables that are accessed most frequently
RichPlus	richPlus	Contains additional RICH variables that are accessed less frequently
UnbExtHitBase	extHitBase	Contains basic variables for unbiased external hits
MCTruthBase	mcTruthBase	Contains basic MC truth variables that are accessed most frequently
MCTruthPlus	mcTruthPlus	Contains additional MC truth variables that are accessed less frequently

The `Event` class acts as an interface to group and access containers with information from the various subdetectors. This should be provided by the chain class as a transient view of the event information.

Note: Containers are actually made up from two classes. The first one is the one holding all the variables, while the second one adds the "read-on-demand" behavior to the container.

When navigating the doxygen documentation remember to go check the *XXXData* class, where "XXX" is the container Class, and you'll find the description for all the container variables.

1.3.1 The event `Category`

To avoid going through every event every single time you can perform a fast event filtering by looking at the event `Category` mask in the `Header` container.

Note: To check if an event belongs in a given set of categories you can use the `Header::CheckMask` method.

Categories can be combined into a single mask, to check many of them at once

```
// this mask will check for charge=1 according to both tracker and tof
NAIA::Category cat = NAIA::Category::Charge1_Trk | NAIA::Category::Charge1_Tof;

  for (NAIA::Event &event : chain) {
    // check charge with TOF and Tracker
    if (!event.header->CheckMask(cat))
      continue;
```

`CheckMask` will check that *all* categories are present in the event. If you want to perform the check in **or** rather than **and** you can use the `MathAnyBit` free function

```
// this mask will check for charge=1 according to both tracker or tof
NAIA::Category cat = NAIA::Category::Charge1_Trk | NAIA::Category::Charge1_Tof;

  for (NAIA::Event &event : chain) {
    // check charge with TOF or Tracker
    if (!NAIA::MatchAnyBit(event.header->Mask(), cat))
      continue;
```

(n.b: the `CheckMask` method uses the `MatchAllBits` free function)

1.4 Skimming

Ntuple skimming is very easy to do in NAIA. You can setup a new file for your skimmed ntuples by calling `NAIAChain::CreateSkimTree` and specifying the name of the output root file (see doxygen).

`CreateSkimTree` will return a `SkimTreeHandle` object. As the name implies the `SkimTreeHandle` is a handle to the new tree, you can call `Fill` on it to save a given event in the new tree.

When you're done processing you can call `Write` on the `SkimTreeHandle` to write the resulting `TTree` on the output root file.

Note: If you don't want to write out some containers (in case you know you won't need them and want to save some space) you can pass a semicolon-separated list of containers to exclude as the second argument of `CreateSkimHandle`.

1.5 Examples

NAIA ships with a few examples to help you getting started

All examples show the steps needed to compile/run a simple executable that loops over a `NAIAChain`

1.5.1 CMake

Warning: This is the recommended way to go

To compile just run

```
mkdir build
cd build
cmake .. -DNAIA_DIR=/path/to/your/naia/install/cmake
make
```

If you take a look at the included `CMakeLists.txt` you'll notice that the only two lines needed to link against the NAIA libraries are:

```
find_package(NAIA REQUIRED)
# ...
target_link_libraries(main PUBLIC NAIA::NAIAChain)
```

this is because NAIA internally defines everything that is needed in terms of targets. The `NAIA::NAIAChain` target internally knows all the include paths, preprocessor macros, library paths, libraries that it needs so that CMake can propagate these requirements to all targets linking against `NAIA::NAIAChain`.

Note: This also means that if in the future these requirements will change you don't have to adapt the build of your project.

1.5.2 Makefile

To compile you need to update the `NAIA_DIR` variable inside the `Makefile` and then you can just call `make`. Remember to add include paths/libraries if needed or if something changes in the NAIA project.

1.5.3 ROOT macros

Before running the examples as a ROOT macros you need to either load the `load.C` macro beforehand

```
root load.C main.cpp
```

or add the content of `load.C` to your `.rootlogon.C`.

Note: Also in this case you have to update the value of `naia_dir` inside of `load.C` and keep track of changes in the upstream NAIA project.

RDataFrame

Note: NB: this mode is not particularly tested, and usage of containers is slightly different

However, it is extremely cool

This example shows how to plot one histogram on NAIA events applying some simple selections, using the `RDataFrame` approach. There are a few caveats when using this approach:

- You don't use `NAIAChain`, instead you have to create the `RDataFrame` object reading the original tree from file, or creating a traditional `TChain`. How this ties with the `RTIInfo` and `FileInfo` trees is to be investigated.
- You have to work with the "Data" container classes, without the "read-on-demand" part. `RDataFrame` is supposed to take care of the rest by itself.
- You have to use the correct branch name in all the operations, which should be the same as the corresponding container "Data" class.

Simple macro

Note: NB: this mode is not particularly tested, and generally discouraged

This example shows how to loop on a `NAIAChain` from a root macro. It is identical to the simple CMake and Makefile examples.