ROOT und C/C++ Referenz für den Praktikumsversuch "Bestimmung der Myonenlebensdauer"

Achim Gütlein

12. Januar 2017

Inhaltsverzeichnis

1 Nützliche Links **2**

2 ROOT Referenz **3**
2.1 TFile . . . 3
2.2 TTree . . . 3
2.3 TCanvas . . . 4
2.4 TH1D . . . 4
2.5 TGraph . . . 6
2.6 TF1 . . . 6
2.7 Fitten . . . 7
2.8 TRandom3 . . . 8
2.9 TMatrixD . . . 8

3 C++ Referenz **9**
3.1 Variablen . . . 9
3.2 Schleifen . . . 10
3.3 Bedingungen . . . 11
3.4 Zeiger (Pointer) . . . 11
3.5 Funktionen . . . 12
3.6 Klassen . . . 13
3.7 Textausgabe . . . 13

1 Nützliche Links

- `root.cern.ch`
  ROOT Homepage
- `https://root.cern.ch/drupal/content/downloading-root`
  ROOT herunterladen und instalieren
- `https://root.cern.ch/doc/master/index.html`
  Dokumentation aller ROOT Klassen
- `https://root.cern.ch/guides/primer`
  Kurze Einführung in ROOT
- `https://root.cern.ch/root-user-guides-and-manuals`
  ROOT Benutzerhandbücher
- `www.cplusplus.com`
  Ausführliche C/C++ Reference mit allen Standardbibliotheken

2 ROOT Referenz

2.1 TFile

- ROOT File öffnen

```
TFile* file = new TFile("FileName.root", "options");
```

- Ist das ROOT File offen?

```
if (!file)
{
   TuDasWennDasFileNichtOffenIst;
}
```

- Objekte laden

```
TTree* tree = (TTree*) file->Get("DataTree");
```

- Objekte speichern

```
tree->Write();
```

2.2 TTree

- Tree erzeugen

```
TTree* tree = new TTree("Name", "Title");
```

- Branch erzeugen

```
double variable;
tree->Branch("BranchName", &variable);
```

- Allen Branches einen Eintrag hinzufügen

```
tree->Fill();
```

- Die Anzahl der Einträge abfragen

```
tree->GetEntries();
```

- Einen existierenden Branch laden

```
double variable;
tree->SetBranchAddress("BranchName", &variable);
```

- Einen Eintrag laden

```
tree->GetEntry(long i);
```

- Einen Tree aus einer formatierten ASCII Datei füllen

```
tree->ReadFile("fileName", "BranchList");
```

  Die Branchliste besteht aus den gewünschten Branchnamen für jede Spalte und den Datentypen für jeden Branch. Die einzelnen Branches werden durch ':' getrennt. Eine Branchbeshreibung besteht aus dem Branchnamen gefolgt von einem '/' und dem Datentyp:

```
tree->ReadFile("MuonData.txt", "Time/D:ClockCycles/I");
```

Die folgenden Datentypen sind verfügbar:

- C: Ein string der durch den Null-character terminiert wird (`char*`)
- B: eine 8 bit lange ganze Zahl mit Vorzeichen (`char`)
- b: eine 8 bit lange ganze Zahl ohne Vorzeichen (`unsigned char`)
- S: eine 16 bit lange ganze Zahl mit Vorzeichen (`short`)
- s: eine 16 bit lange ganze Zahl ohne Vorzeichen (`unsigned short`)
- I: eine 32 bit lange ganze Zahl mit Vorzeichen (`int`)
- i: eine 32 bit lange ganze Zahl ohne Vorzeichen (`unsigned int`)
- L: eine 64 bit lange ganze Zahl mit Vorzeichen (`long`)
- l: eine 64 bit lange ganze Zahl mit Vorzeichen (`unsigned long`)
- F: eine 32 bit lange Fliekommazahl (`float`)
- D: eine 64 bit lange Fliekommazahl (`double`)
- O: (der Buchstabe 'O', keine Null) eine Bool'sche Variable (`bool`)

2.3 TCanvas

- Ein Canvas erzeugen

```
TCanvas* canvas = new TCanvas("Name", "Title");
```

- Logarithmische Axen

```
canvas->SetLogx(); // x-axis
canvas->SetLogy(); // y-axis
```

- Ein Gitter zeichen

```
canvas->SetGrid();
```

- Ein Canvas aktiv machen

```
canvas->cd();
```

2.4 TH1D

- Ein Histogramm mit gleichbleibender Binbreite erzeugen

```
TH1D* histo = new TH1D("Name", "Title;x-Title;y-Title", int
    nBins, double xMin, double xMax);
```

- Ein Histogramm mit beliebiger Binbreite erzeugen

```
double* lowEdges = new double[nBins+1]
lowEdges[0] = xMin;
...
TH1D* histo = new TH1D("Name", "Title;x-Title;y-Title", int
    nBins, double* lowEdges);
```

- Nummerierung von Bins:

  - Bin 0: Underflow
  - Bin 1: Erstes Bin mit unterer Grenze xMin
  - Bin nBins: Letztes Bin mit oberer Grenze xMax
  - Bin nBins+1: Overflow

- Einen Wert in ein Histogramm füllen

```
histo->Fill(double value);
```

- Einen Bineintrag abfragen

```
histo->GetBinContent(int binNumber);
```

- Einen Bineintrag setzen

```
histo->SetBinContent(int binNumber);
```

- Die Binmitte ausgeben

```
histo->GetBinCenter(int binNumber);
```

- Die untere Grenze eines Bins ausgeben

```
histo->GetBinLowEdge(int binNumber);
```

- Die obere Grenze eines Bins ausgeben (die untere Grenze des nächsten Bins)

```
histo->GetBinLowEdge(int binNumber + 1);
```

- Minimum und Maximum der Y-Achse zum Zeichen setzen

```
histo->SetMinimum(double yMin);
histo->SetMaximum(double yMax);
```

- Das Statistikfenster an/abschalten

```
histo->SetStats(kTRUE); // statistics on
histo->SetStats(kFALSE); // statistics off
```

- Die Linienfarbe ändern

```
histo->SetLineColor(int color);
```

- Die Linienbreite ändern

```
histo->SetLineWidth(int width);
```

- Ein Histogramm zeichnen

```
histo->Draw("options");
```

  Optionen:

  - axis: Zeichent nur die Axsen des Histogramms (z.B. um Axen für einen Graphen (Abschnitt 2.5) zu erzeugen)
  - ...
  - same: Fügt das Histogramm dem bereits bestehenden Canvas hinzu

2.5 TGraph

- Einen Graph erzeugen

```
double* xValues = new double[nPoints];
double* yValues = new double[nPoints];

xValues[0] = xValue0;
yValues[0] = yValue0;
...

TGraph* graph = new TGraph(nPoints, xValues, yValues);
```

- Marker Style und Farbe ändern

```
graph->SetMarkerStyle(int style); // marker style
graph->SetMarkerColor(int color); // marker color
```

- Einen Graph Zeichen

```
graph->Draw("options");
```

  Optionen:

  - P: Zeichnet Marker im gewähltem Style
  - *: Zeichnet * als Marker (unabhängig vom gewähltem Style)
  - L: Verbindet alle Punkte durch Geraden
  - C: Zeichnet eine geglättete Linie durch alle Punkte
  - A: Erzeugt Axen für den Graph
  - ...
  - same: Fügt den Graph dem bereits bestehenden Canvas hinzu

2.6 TF1

- Eine Funktion erzeugen

```
TF1* func = new TF1("Name", "Formula", xMin, xMax);
TF1* func = new TF1("Name", "[0]*x*x + [1]*x + [2]", xMin,
    xMax); // Parabola
```

- Parameter setzen

```
func->SetParameter(0, value); // Set one parameter
func->SetParameters(value0, value1, value2, ...); // Set all
    parameters at once
```

- Linienfarbe und Linienbreite einstellen

```
func->SetLineColor(int color);
func->SetLineWidth(int width);
```

- Eine Funktion zeichnen

```
func->Draw("options");
```

  Optionen:

  - ...
  - same: Fügt die Funktion dem bereits bestehenden Canvas hinzu

- Anzahl der Stützstellen ändern

```
func->SetNpx(int value); // maximum 100000!
```

2.7 Fitten

- Eine Funktion an ein Histogram fitten:

```
hist->Fit(func, "opt", "draw opt", xMin, xMax);
hist->Fit("func", "opt", "draw opt", xMin, xMax);
hist->Fit(func, "LRI");
```

- Eine Funktion an einen Graphen fitten (Curvefit):

```
graph->Fit(func, "RI");
```

- Beim Fitten an einen Graphen werden alle Fehler auf 1 gesetzt, wenn die Fehler nicht explizit angegeben sind (→ `TGraphErrors`, `TGraphAsymmErrors`)
- Wichtige Fitoptionen:
  - `"R"` Im Definitionsbereich der Fitfunktion fitten
  - `"L"` Log-Likelihood-Fit
  - `"I"` Integral der Funktion anstatt des Wertes in der Mitte des Bins verwenden
  - `"W"` Alle Fehler werden auf 1 gesetzt; leere Bins werden ignoriert
  - `"WW"` Alle Fehler werden auf 1 gesetzt auch fr leere Bins
  - `"Q"` Fitergebnisse nicht auf der Konsole ausgeben
  - `"V"` Alle verfgbaren Informationen auf der Konsole ausgeben
  - `"O"` Fitfunktion nicht zeichnen
- Beim Fitten eines Histograms werden für einen χ^2-Fit als Fehler die Wurzel der Bineinträge verwendet. Leere Bins werden bei diesem Fit ignoriert.
- Beim Log-Likelihood-Fit wird angenommen, dass die Bineintrge/ Datenpunkte Poisson verteilt sind.
- Um andere Verteilungen zu fitten kann man ROOT sagen, welche Funktion minimiert werden soll (z.B. Likelihood-funktion mit Binomialverteilung)
- Die Fitergebnisse können direkt von der Fitfunktion ausgelesen werden. Den gefitteten Wert eines Parameters und den entsprechenden (1σ) Fehler erhält man mittels:

```
func->GetParameter(0)
func->GetParError(0)
```

  Das χ^2 und die Anzahl der Freiheitsgrade (Number of Degrees of Freedom, NDF) erhält man mittels

```
func->GetChisquare();
func->GetNDF();
```

2.8 TRandom3

- Einen Zufallsgenerator erzeugen

```
TRandom3 rnd;
```

- Einen zufälligen Seed wählen

```
rnd.SetSeed(0);
```

- Eine gleichverteilte Zufallszahl zwischen 0 und 1 erhalten

```
rnd.Uniform();
```

- Eine normalverteilte Zufallszahl

```
rnd.Gaus(double mean, double sigma);
```

- Eine Poisson verteile Zufallszahl

```
rnd.Poisson(double mean);
```

- Eine exponentiel verteilte Zufallszahl ($f(t) = \frac{1}{\tau}e^{-\frac{t}{\tau}}$)

```
rnd.Exp(double tau);
```

2.9 TMatrixD

- Eine Matrix erzeugen

```
TMatrixD mat(int rows, int columns);
```

- Eine Matrix mit Werten füllen

```
mat(0, 0) = value00;
```

- Eine Matrix Invertieren

```
mat.Invert();
```

- Eine Matrix ausgeben

```
mat.Print();
```

- Auf die Werte zugreifen

```
mat(0, 0);
```

3 C++ Referenz

3.1 Variablen

- Eine Variable deklarieren

```
double variable;
double variable = 0.;
```

- Basisdatentypen

Datentyp	Beschreibung	Größe [Byte]
`short`	Ganzzahl	2
`int`	Ganzzahl	4
`long`	Ganzzahl	8
`float`	Fließkommazahl	4
`double`	Fließkommazahl	8
`bool`	Boolsche Variable	1
`char`	1 ASCII Zeichen	1

- Rechenoperatoren

Operator	Beschreibung
`=`	Zuweisung
`+`	Addition
`-`	Subtraktion
`*`	Multiplikation
`/`	Division
`%`	Modulo

- Vorsicht bei Division von ganzen Zahlen:

```
1/3 = 0
1./3. = 1/3. = 1./3 = 0.3333333
```

- Rechenreihenfolge: Punkt vor Strich, ansonsten der Reihe nach. Es ist möglich runde Klammern zu verwenden.

- Erweiterte Operatoren

Operator	Verwendung	Beschreibung
`++`	`i++`	`i = i + 1`
`--`	`i++`	`i = i - 1`
`+=`	`a += b`	`a = a + b`
`-=`	`a -= b`	`a = a - b`
`*=`	`a *= b`	`a = a * b`
`/=`	`a /= b`	`a = a / b`

- Vergleichsoperatoren

Operator	Beschreibung
>	Größer
<	Kleiner
==	Ist gleich
!=	Ungleich
>=	Größer gleich
<=	Kleiner gleich

- Logische Operatoren

Operator	Beschreibung
&&	Und
||	Oder
!	Nicht

- Gültigkeitsbereiche von Variablen

```
double a; // global variables should be avoided, if possible

void Function(double b)
{
        cout << b << endl; // ok

        for (int i=0; i<b; i++)
        {
                b += i; // ok
        }

        cout << i; // error!
        cout << a; // ok
}

cout << b << endl; // error
cout << a; // ok
```

3.2 Schleifen

- for-Schleife

```
for (int i=0; i<4; i++)
{
        cout << i << ", "; // Ausgabe: 0, 1, 2, 3,
}
```

- while-Schleife

```
while(condition == true)
{
        DoSomething;
}
```

- Steuerbefehle:
  - `continue` - Es wird sofort der nächste Schleifendurchlauf begonnen
  - `break` - Die Schleife wird sofort verlassen

3.3 Bedingungen

- if Bedingung

```
if (condition == true) DoSomething;

if (condition == true)
{
        DoOneThing;
        AndAnother;
}
```

- if und else

```
if (condition == true)
{
        DoSomething;
}
else
{
        DoSomethingElse;
}
```

- Komplexere Strukturen

```
if (condition = true) DoSomething;
else if (otherCondition == true) DoSomethingElse;
...
else DoEvenMore;
```

3.4 Zeiger (Pointer)

- Einen Zeiger deklarieren

```
double* p; // pointer to a double
```

- Die Speicheradresse einer Variablen erhalten

```
double a;
double* p = &a;
```

- Auf den Inhalt einer Speicheradresse zugreifen

```
double a = 2;
double* p = &a;
*p = 4;
cout << a; // Ausgabe: 4
```

- Mit `new` Speicher reservieren

```
TH1D* hist = new TH1D("Name", ...);
double* a = new double;
```

- Mit `delete` reservierten Speicher freigeben

```
        TH1D* hist = new TH1D("Name", ...);
        ...
        delete hist;
```

- Arrays mit beliebiger Länge erzeugen

```
int nSamples = 10;
double* a = new double[nSamples];
a[0] = 0.; // First element set to 0.;
a[nSamples -1] = -10.; // Last element set to -10.
...
delete[] a; // deletes the array
```

- Segmentation Fault/ Segmentation violation
  Zeiger deklariert, aber nicht initialisiert:

```
double* p; // Pointer declared but not initialized!
*p = 5; // -> Segmentation fault

TH1D* hist;
hist->GetBinContent(0); // -> Segmentation fault
```

  Versuch auf nicht reservierten Speicher zuzugreifen

```
double* a = new double[5];
a[5] = 6; // -> Segmentation fault (Last element has index 4!)
```

  Zeiger zeigt auf bereits wieder freigegebenen Speicher:

```
double* a;
if (condition)
{
        double b = 10;
        a = &b;
}

*a = 5; // -> Segmentation fault (b is out of scope!)
```

3.5 Funktionen

- Funktion definieren

```
<returnType> FunctionName (input data list)
{
        DoSomeThing;
}
```

- Unterschiedliche Typen für die Rückgabe einer Funktion
- Rückgabetyp `void` wenn nichts zurrückgegeben werden soll:

```
void PrintValue(double value)
{
        cout << "Value = " << value << endl;
}

double Square(double value)
{
        return value * value;
}
```

- Zeiger verwenden um Eingangsdaten zu verändern

```
void swap(double* a, double* b)
{
        double temp = *a;
        *a = *b;
        *b = temp;
}
```

- Funktionen verwenden

```
aSquare = Square(a);
Print(aSquare);
swap(&a, &aSquare);
```

3.6 Klassen

- Klassen sind komplexe Datentypen (z.B.: double-array → Vector-Klasse)
- Zugriff auf öffentliche Methoden (= Funktionen) und Variablen

```
TH1D hist("Name", ...);
hist.GetBinContent(1);

TH1D* hist = new TH1D("Name", ...);
hist->GetBinContent(1);
```

3.7 Textausgabe

- Textausgabe mit `cout`

```
cout << "Ein Text"; // Ein Text
double a = 10.5;
cout << a; // 10.5
cout << "a = " << a << "cm"; // a = 10cm
```

- Kein Zeilenumbruch bei neuem `cout`

```
cout << "Erste Zeile";
cout << "Zweite Zeile";
// Erste ZeileZweite Zeile
```

- Neue Zeile mit `endl`

```
cout << "erste Zeile" << endl << "Zweite Zeile";
// erste Zeile
// Zweite Zeile
```

- Um Problemen mit Zeilen umbrüchen vorzubeugen, immer `endl` verwende

```
cout << "Ein Text" << variable << endl;
```