Einführung in ROOT und C++

Achim Gütlein

Institut für Hochenergiephysik
Österreichische Akademie der Wissenschaften

13. Januar 2017

Ein bisschen C++

Variablen

- Variablen definieren/deklarieren und initialisieren

```
double a = 1.;
int b, c;
short n1 = n2 = n3 = 0;
TRandom3 rnd;
```

- Rechenoperationen

```
a = b + c;
i++;
mean /= nSamples;
```

- Vorsicht bei Division von ganzen Zahlen:

```
1/3 = 0;
1./3. = 1/3. = 1./3 = 0.33333333333
```

Textausgabe

- Es gibt mehrere Möglichkeiten Text auszugeben
- Eine Möglichkeit bietet cout

```
double a = 0.5;
cout << "Text, a = " << a << ", Zahl: " << 10*0.5;
// Ausgabe: Text, a = 0.5, Zahl: 5.0
```

- Bei mehrmaligem Aufrufen von cout wird kein Zeilenumbruch ausgeführt

```
cout << "1. Zeile";
cout << "2. Zeile";
// Ausgabe: 1. Zeile2. Zeile
```

- endl für Zeilenumbruch verwenden.

```
cout << "1. Zeile" << endl << "2. Zeile" << endl;
cout << "Zahl: " << 42 << endl;
```

Kontrollstrukturen - Bedingungen

- Bedingungen werden sowohl für Schleifen als auch die if-Anweisung verwendet.
- Eine Bedingung ist eine boolsche Variable. Sie ist also wahr (`true`) oder falsch (`false`)

```
bool useSimData = true;
```

- Auch Ergebnisse von Vergleichsoperatoren können als Bedingungen verwendet werden

```
a == 0
b<0 || b>=10
```

- Auch die Basisdatentypen und Zeiger können als Bedingungen interpretiert werden.

→ 0 ist `false`; alle anderen Werte sind `true`

```
!histo true, wenn histo nicht initialisiert
```

Kontrollstrukturen - if Anweisung

- Struktur:

```
if (bedingung) Anweisung;
else AndereAnweisung;
```

- Wenn `bedingung` wahr ist, dann wird `Anweisung` ausgeführt, sonst `AndereAnweisung`
- Die Anweisungen können einzelne Anweisungen (a=0) oder Anweisungsblöcke sein

```
if (bedingung)
{
    Anweisung1;
    Anweisung2;
}
```

- Auch komplexere Strukturen sind möglich

```
if (bedigung1) Anweisung1;
else if (bedingung2) Anweisung2;
else Anweisung3;
```

Kontrollstrukturen - for-Schleifen

→ Wenn bereits bekannt ist wie oft die Anweisung wiederholt werden muß, kann die for-Schleife verwendet werden

```
for (Initialisierung; Abbruchbedingung; Anweisung)
{
    Anweisung(i);
}
```

- Anweisungsblock wird wiederholt, bis Abbruchbedingung (`i<10`) erfüllt
- Nach jeder Wiederholung wird die Anweisung `i++` ausgeführt
- Wird z.B. verwendet um die Werte eines Arrays zu maipulieren

```
for (int i=0; i<3; i++)
{
    t[i] += randomNumber;
}
```

Kontrollstrukturen - while-Schleife

→ Wenn nicht bekannt ist wie oft die Anweisung wiederholt werden muß, kann die while-Schleife verwendet werden

```
while(bedingung)
{
    Anweisungen;
}
```

- Anweisungsblock wird wiederholt, solange die Bedingung wahr ist.
- Wird z.B. verwendet um Daten aus einer Datei zu lesen:

→ Wie oft man lesen muß weiß man nämlich erst wenn man das Ende der Datei erreicht hat.

```
while(!file.eof())
{
    ReadLine();
}
```

Kontrollstrukturen - Schleifen

- Bei ineinander verschachtelten for-Schleifen sollte man auf die Bezeichnung der Variablen achten

```
for (int i=0; i<3; i++)
{
    a[i] = 0;

    for (int j=0; j<3; j++)
    {
        a[i] += b[i][j]
    }
}
```

- Mit dem Befehl `continue` wird sofort der nächste Schleifendurchlauf gestartet
- Mit `break` wird die Schleife sofort verlassen

Funktionen

- Ein Anweisungsbock mit Namen, Eingangsvariablen und Rückgabewert

```
<Type> Name (Var1, Var2, ...)
```

- Rückgabetyp `void` für keine Rückgabe

```
void Print(double a) { cout << a << endl; }
```

- Variablen in Eingangsliste können nicht verändert werden:

```
void swap (double a, double b) // a = 2, b = 3
{
    double temp = a; // temp = 2
    a = b;           // a = 3
    b = temp;        // b = 2
}
double a = 2; double b = 3;
swap (a, b); // -> a = 2, b = 3!
```

→ `swap(a, b)` ist identisch zu `swap(2, 3)`

- Funktionen bekommen nur den Wert der Variablen übergeben

Zeiger

- Zeiger sind spezielle Variablen, deren Werte Speicheradressen von anderen Variable sind.
- Zeiger definieren:

```
double* p;
TH1D* histo = 0;
```

- Zeigern eine Speicheradresse zuweisen

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

- Den gespeicherten Wert einer Speicheradresse verändern

```
*p = 8;
```

- Speicher reservieren; Dynamische Arrays

```
TH1D* histo = new TH1D(...)
double* xValues = new double[nPoints];
```

- Speicher freigeben ($\rightarrow$ Memory leaks vermeiden)

```
delete histo;
delete[] xValues;
```

Zeiger und Funktionen

- Zeiger haben als Wert einen Speicheradresse
- Der Inhalt von Speicheradressen kann verändert werden

→ Zeiger verwenden um in Funktionen den Wert von Variablen zu ändern

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

double a = 2;
double b = 3;

swap(&a, &b); // -> a = 3, b = 2
```

Klassen

- Klassen sind komplexe Datentypen

```
double mat1[3][3];
TMatrixD mat2(3, 3);
```

- Klassen kombinieren Variablen und Funktionen

```
mat2.Invert();
```

- Man kann nur auf öffentliche Variablen und Funktionen zugreifen

```
TRandom3 rnd;
rnd.Uniform();

TH1D* histo = new TH1D(...);
histo->GetBinContent(1);
```

- Variablen deren Typ eine Klasse ist werden als Objekte oder Instanzen der Klasse bezeichnet

ROOT Crashkurs

Was ist ROOT?

- Datenanalyse Framework
- Entwickelt am CERN
- Mehrere C++ Bibliotheken
- Alles vom Taschenrechner bis zur LHC Analyse
- Datenbank, Plots, Fits, Statistik, ...
- Kommandozeileninterpreter
- Makros (einzelne C++ Funktionen)
- Standaloneprogramme

→ root.cern.ch

ROOT Objekte

- Den meisten ROOT Objekte kann ein Name gegeben werden

```
TH1D* histo = new TH1D("Name", ...);
TF1* func = new TF1("Name", ...);
```

- Bei ROOT kann man auf Objekte mit diesem Namen zugreifen

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

- Wenn ein neues Objekt mit dem selben Namen erzeugt wird, kann das existierende Objekt verloren gehen

⇒ Unterschiedliche Namen wählen

- Es ist möglich eigene Klassen zu erstellen und sie mit ROOT zu verbinden

Kommandozeileninterpreter

- Start mit `root` bzw. `root -l` (kein Startbild)
- Versteht C++ Befehle
- Kennt alle ROOT Klassen
- Kennt einige Standard C++ Bibliotheken
- Bis Version 5.34: Versucht Fehler zu korrigieren
- Ausführung von einzelnen Befehlen
- Ausführung von Makros (C++ Dateien mit einer oder mehreren Funktionen)
- Ausführen eines Makros mit `.x MacroName.C`:
  - → Die Funktion `MacroName` wird ausgeführt
- Laden eines Makros mit `.L MacroName.C`:
  - Alle Funktionen werden geladen
  - Der Interpreter kennt alle Funktionen
  - → Die Funktionen können auf der Kommandozeile benutzt werden.

TFile

- Schnittstelle zu ROOT Files
- Alle ROOT Objekte können in ROOT Files gespeichert werden

```
RootObject->Write();
```

- Objekte können mit ihrem Namen geladen werden

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

- Erzeugen eines ROOT Files

```
TFile* file = new TFile("File.root", "recreate");
TFile* file = new TFile("File.root");
TFile* file = new TFile("File.root", "update");
```

- Check ob ROOT File geöffnet werden konnte

```
if (!file)
{
        cout << "File not found!" << endl;
        return;
}
```

TTree

- ROOT Datenbanksystem
- Einen Tree erzeugen

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

- Tree wird automatisch mit dem aktuellem ROOT File verknpft
- Trees ohne ROOT File können zu Problemen führen
- Wie eine Tabelle organisiert
- Spalten werden als Branch bezeichnet
- Branches können alle Datentypen oder Klassen speichern, die ROOT bekannt sind
- Einen neuen Branch erzeugen

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

TH1D* hist;
tree->Branch("BranchName", &hist);
```

TTree

- Einen bestehenden Branch laden

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

TH1D* hist;
tree->SetBranchAddress("BranchName", &hist);
```

- Einen Tree füllen → der Tabelle eine neue Zeile hinzufügen

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

→ Für alle Branches werden die Werte ihrer assoziierten Variablen gespeichert

- Die Anzahl der Einträge (Zeilen der Tabelle) abfragen

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

- Einen Eintrag laden

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

→ Die assozierten Variablen aller Branches werden mit den gespeicherten Werten gefüllt.

TH1D

- Klasse für 1D Histogramme mit einer `double` pro bin
- Histogram mit konstanter Binbreite erzeugen

```
TH1D* hist = new TH1D("Name", "Titel;xAchse;yAchse
    ", nBins, xMin, xMax);
```

- Histogram mit beliebiger Binbreite erzeugen

```
double* lowEdges = new double[nBins+1];
lowEdges[0] = xMin;
...
lowEdges[nBins+1] = xMax;

TH1D* hist = new TH1D("Name", "Title", nBins,
    lowEdges);
```

- Neueun Eintrag hinzufügen

```
hist->Fill(value);
```

TH1D

- Einen Bininhalt abfragen

```
hist->GetBinContent(i);
```

- Nummerierung der Bins:
  - 0 → Underflow
  - 1 Untere Grenze (eingeschloßen) `xMin`
  - ...
  - `nBins` → Obere Grenze (ausgeschloßen) `xMax`
  - `nBins + 1` → Overflow
- Statistikfenster ausblenden

```
hist->SetStats(0);
```

- Ein Histogram zeichnen

```
hist->Draw();
hist->Draw("same");
hist->Draw("axis");
```

- Limits für die Y-Achse ändern

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

TGraph

- Eine Menge von Punkten
- Einen Graph erzeugen

```
double* xValues = new double[nPoints];
double* yValues = new double[nPoints];
xValues[0] = ...;
yValues[0] = ...;
...
TGraph* graph = new TGraph(nPoints, xValues,
    yValues);
```

- Einen Graph zeichnen

```
graph->Draw("P same");
graph->Draw("PL same");
graph->Draw("A *");
```

- Für eigene Achsenlimits ein Histogramm mit den entsprechenden Limits erstellen

```
hTemp->Draw("axis");
graph->Draw("P same");
```

TF1

- Eine Klasse für beliebige 1D Funktionen
- Ein TF1 Objekt erzeugen

```
TF1* func = new TF1("Name", "Formula", xMin, xMax)
```

- Beispiele mit einer Parabel und einer Exponentialfunktion

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

- Werte für Parameter ([0], [1], ...) setzen

```
func->SetParameter(0, value);
func->SetParameters(value0, value1, ...);
```

- Eine Funktion zeichnen

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

- Anzahl der Stützstellen ändern

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

Fit

- Eine Funktion an ein Histogram fitten:

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

- 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
- 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)

TCanvas

- Ein Fenster in das unterschiedliche Objekte gezeichnet werden können
- Ein Canvas erzeugen

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

- Im Menü unter „View“ lässt sich ein Editor öffnen mit dem man die Zeichenoptionen anpassen kann
- Ein Canvas zum aktiven Canvas machen

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

- Logarithmische Achsen

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

Zeichenoptionen

- Bei ROOT Objekten wie TH1D, TGraph, TF1, ... lassen sich die Zeichenoptionen anpassen.
- Anpassen der Linienfarbe und -breite

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

- Die ROOT Farbpallete anzeigen

```
TCanvas* canvas = new TCanvas("c", "Colors");
canvas->DrawColorTable();
```

- Die Marker von Graphen anpassen

```
graph->SetMarkerStyle(int style);
graph->SetMarkerColor(int style);
```

- Die verfügbaren Markerstyles anzeigen

```
TMarker marker;
marker.DisplayMarkerTypes();
```

TRandom3

- Zufallszahlengenerator von ROOT

```
TRandom3 rnd;
```

- Eine gleichverteile Zufallszahl aus $[0, 1[$ erhalten

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

- Einen zufälligen Seed einstellen

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

- Wenn kein zufälliger Seed eingestellt wird, ist die Folge der Zufallszahlen immer identisch.

Andere Verteilungsfunktionen

- Aus einer gleichverteilten Zufallszahl ($U \in [0,1]$, $g(U) = 1$) lässt sich eine andere Zufallszahl x bestimmen die nach einer Funktion $f(x)$ verteilt ist:

$$x = F^{-1}(U)$$

- Eine exponentiel verteilte Zufallszahl erhalten wir also so:

$$\begin{aligned} f(t) &= \frac{1}{\tau} e^{-\frac{t}{\tau}} \\ F(t) &= \int_0^t f(t')dt' = 1 - e^{-\frac{t}{\tau}} \\ F^{-1}(U) &= -\tau \ln(1-U) \end{aligned}$$