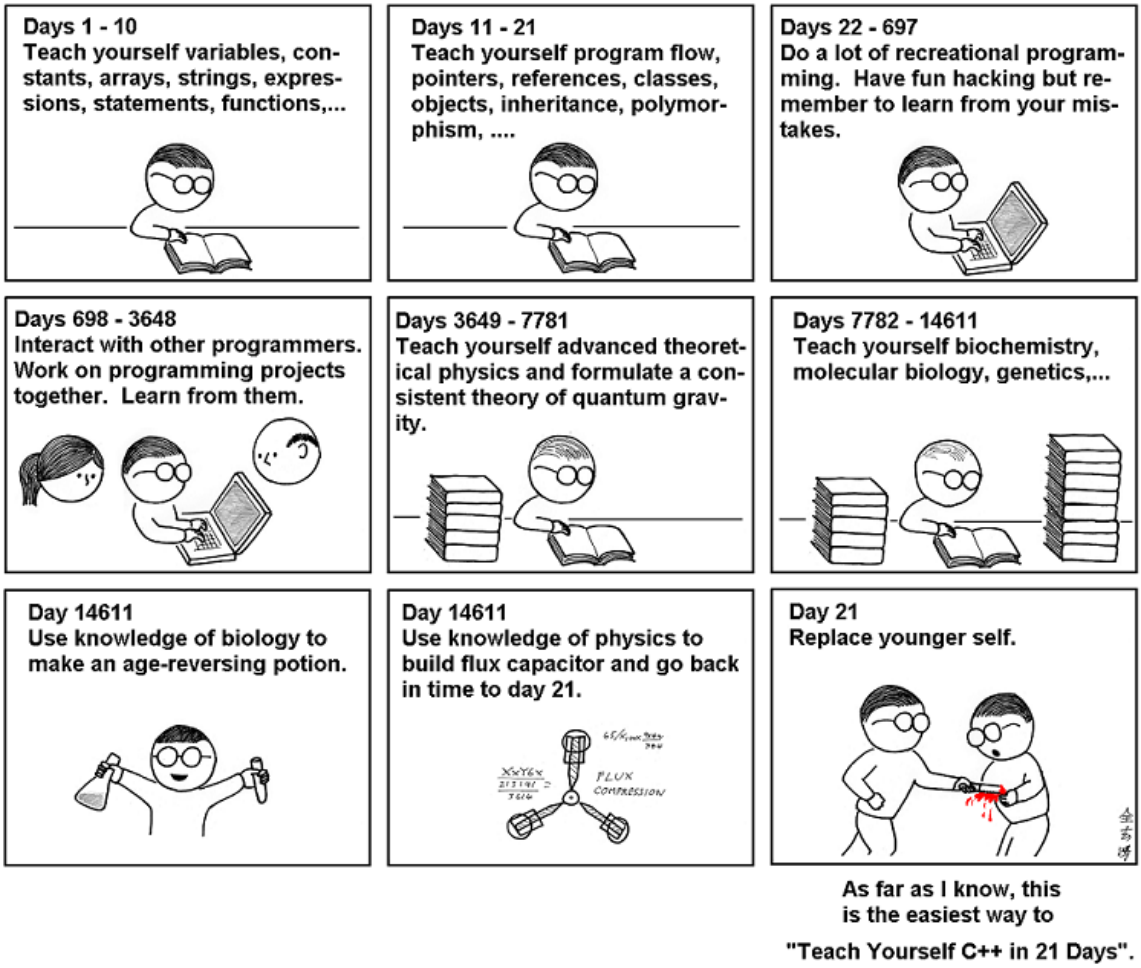
Inhaltsverzeichnis


Days 1 - 10
Teach yourself variables, constants, arrays, strings, expressions, statements, functions,...
Days 11 - 21
Teach yourself program flow, pointers, references, classes, objects, inheritance, polymorphism, ...
Days 22 - 697
Do a lot of recreational programming. Have fun hacking but remember to learn from your mistakes.
Days 698 - 3648
Interact with other programmers. Work on programming projects together. Learn from them.
Days 3649 - 7781
Teach yourself advanced theoretical physics and formulate a consistent theory of quantum gravity.
Days 7782 - 14611
Teach yourself biochemistry, molecular biology, genetics,...
Day 14611
Use knowledge of biology to make an age-reversing potion.
Day 14611
Use knowledge of physics to build flux capacitor and go back in time to day 21.
FLUX COMPRESSION
Day 21
Replace younger self.
As far as I know, this is the easiest way to
"Teach Yourself C++ in 21 Days".

Teil I

Einführung

Kapitel 1

Computer in der Kernphysik

Computer sind in der Kern- und Teilchenphysik ein allgegenwärtiges Werkzeug, das in praktisch allen Bereichen von Experimenten und auch häufig in der Theorie angewandt wird.

Simulationen werden in der Planung von Experimenten und einzelnen Messungen werden eingesetzt, um physikalische Prozesse und Detektoren zu studieren und so z.B. optimale Detektoren zu entwerfen oder die Machbarkeit von Analysen zu untersuchen.

Datennahme umfasst die Auslese und das Abspeichern der Daten während eines Experimentes. Bei modernen Experimenten müssen Datenströme bis zu mehreren GByte/sec verarbeitet werden.

Rekonstruktion ist die Aufbereitung der Daten, um aus Rohdaten (i.d.R. ADC-Werte der beteiligten Detektoren) relevante physikalische Größen wie Teilchenimpulse oder -energien zu extrahieren. Die Rekonstruktion der Daten einer Strahlzeit kann mehrere Monate auf hunderten oder tausenden CPUs dauern.

Analyse ist die Extraktion von Signalen aus rekonstruierten Daten. Dies ist die Hauptaufgabe in vielen Doktorarbeiten.

Grafische Darstellung x-y-Plots, Histogramme

Aufgrund der riesigen Datenmengen sind Experimente der Kern- und Teilchenphysik oftmals führend in der Einführung neuer Technologien. So wurde für den LHC ein

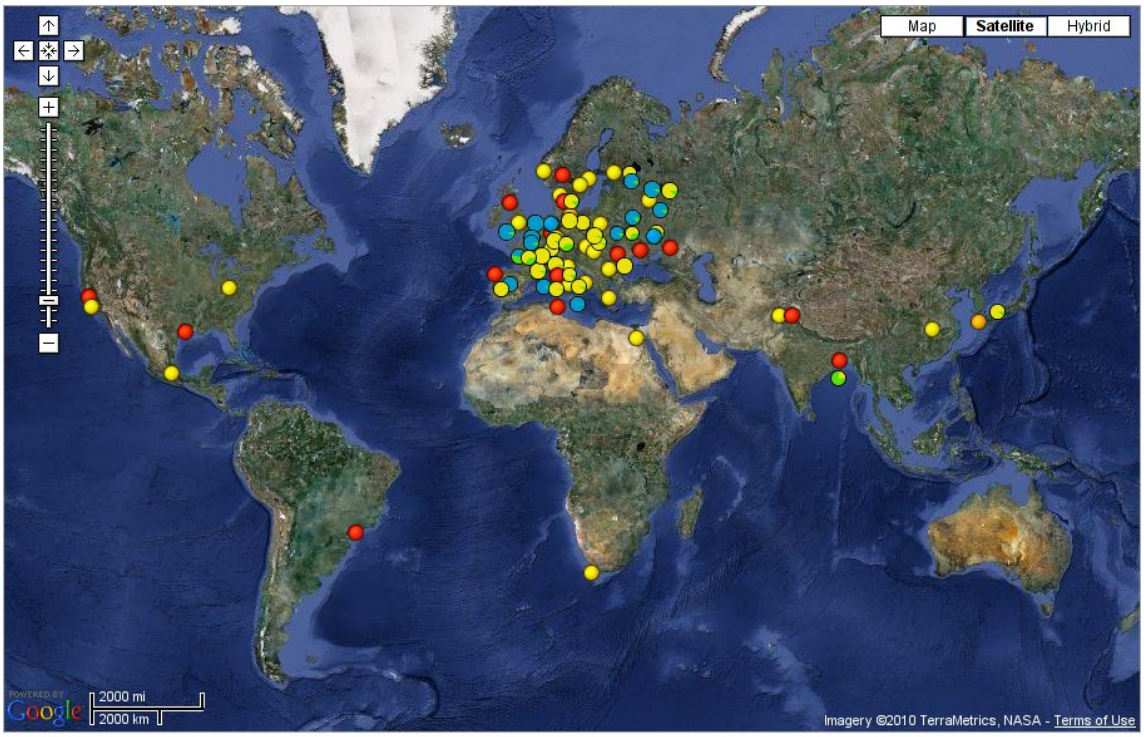


Abbildung 1.1: ALICE Grid: Rechenzentren des ALICE Experimentes

Grid entwickelt, mit dem die Rechenzentren der an den Experimenten beteiligten Insitute weltweit vernetzt sind.

Kapitel 2

Computer-Architektur

2.1 CPU

Die Zentrale Recheneinheit (engl.: Central Processing Unit, CPU) verarbeitet Daten anhand der Befehle in einem Programm.

- Zugriff auf Arbeitsspeicher
- Arithmetik: + - * / (log, sqrt)
- Logik: $=\neq <>$
- Sprünge, bedingte Sprünge
- Ein-/Ausgabe über Hardware-Register

Tyische Geschwindigkeit: Taktfrequenz 2–3 GHz, mehrere Befehle pro Takt und Prozessorkern (Core)

2.2 Arbeitsspeicher

Engl.: Memory, Random Access Memory, RAM

Speicher für Befehle und Daten während der Ausführung eines Programms. Befehle und Daten werden als ganze Zahlen in Speicherstellen abgelegt, die einzeln addressiert werden können.

Typische Geschwindigkeit: Transferrate mehrere GB/s, Latenzen bis zu 100 ns

Cache Kleiner Zwischenspeicher mit sehr schneller Anbindung an CPU. Oft mehrstufig mit Größen von kB bis MB.

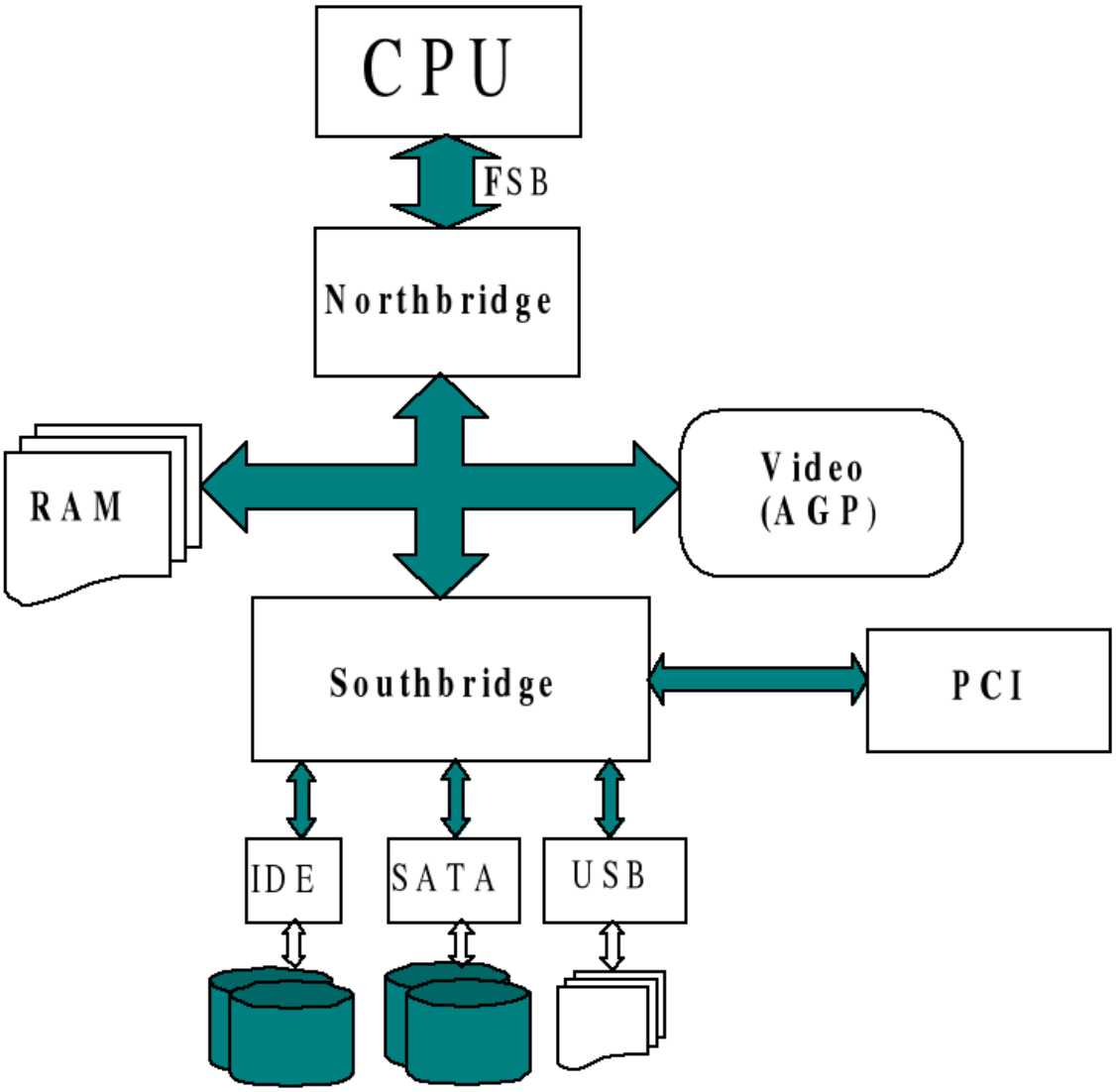


Abbildung 2.1: Aufbau eines Computers

Swap Arbeitsspeicher kann auf Festplatte ausgelagert werden, wenn er nicht benötigt wird. Sehr billig, aber sehr langsam.

2.3 Ein-/Ausgabegeräte

Engl.: Input/Output, I/O
I/O-Geräte werden über Busse angeschlossen: PCI, IDE, SATA, USB

Monitor und Tastatur

Festplatte Speicher für große Datenmengen (bis zu TB), 100 MB/s

Netzwerk Zugriff auf Daten, die auf anderen Rechnern gespeichert sind oder dort produziert wurden. 100 MB/s (Gigabit Ethernet)

DAQ-System Auslese von Messdaten. Limitiert durch Bussystem, mehrere 100 MB/s

Kapitel 3

Programmiersprachen

3.1 Maschinensprache, Interpreter und Compiler

Maschinensprache oder Assembler ist die eigentliche Sprache des Prozessors. Es stehen nur einfachste Befehle zur Verfügung, höhere Funktionen (z.B. I/O) werden vom Betriebssystem zur Verfügung gestellt.

Maschinensprache ist sehr schwer zu schreiben, bietet kaum Programmierhilfen und ist nicht auf andere Prozessoren portabel (z.B. von i386 auf amd64).

Maschinensprache kann—im Prinzip—optimal an einen Computer angepasst werden. Moderne optimierende Compiler erzeugen aber fast immer schnelleren Code.

Höhere Programmiersprachen erlauben eine abstrakte Darstellung eines Algorithmus, die für Menschen verständlicher ist, aber nicht direkt vom Computer ausgeführt werden kann.

Interpreter übersetzen ein Programm in einer höheren Programmiersprache während der Ausführung in Maschinensprache. Ein Programm kann sofort nach dem Schreiben ausgeführt werden, läuft allerdings relativ langsam.

Compiler übersetzen das Programm einmal in Maschinensprache. Das kompilierte Programm kann dann mehrmals ausgeführt werden. Das Programm läuft schneller als in einem Compiler, die Compilation kostet allerdings einmalig Zeit.

3.2 C, C++ und ROOT

Geschichte von C und C++

1969	Start der Entwicklung von C als systemnahe Programmiersprache an den AT&T Bell Labs, basierend auf BCPL und B
1973	Implementation des Unix-Kernels in C
1978	*The C Programming Language* von Brian Kernighan und Dennis Ritchie erscheint (K&R C)
1979	Bjarne Stroustrup startet *C with Classes*, eine objektorientierte Erweiterung von C
1983	Namensänderung von C with Classes in C++
1985	Erster kommerzieller C++-Compiler
1985	*The C++ Programming Language* von Bjarne Stroustrup erscheint
1989	Standardisierung von C durch das American National Standards Institute (ANSI-C)
1999	Erweiterung des C-Standards (C99)
1998	Standardisierung von C++ (C++98)
2003	Erweiterung des C++-Standards (C++03)
2011	Erweiterung des C++-Standards (C++11)

C

- Programmiersprache für Systementwicklung
  - schnell (Compiler)
  - Kontrolle über Hardware, direkter Speicherzugriff
  - kein Speicherschutz
  - "der Programmierer weiß, was er tut"
- Kleiner Sprachkern
- Funktionen
  - Unterprogramme mit Namen
  - Aufruf mit unveränderbaren Argumenten
  - Seiteneffekte (z.B. Bildschirmausgabe)
  - Rückgabe eines Wertes (z.B. Ergebnis einer Berechnung)
- Komplexe Datentypen

  - Zusammenfassen von zusammengehörigen Daten (z.B. Komponenten eines Vektors, Name und Geburtsdatum von Personen)

- Erweiterbarkeit über Bibliotheken
  - Standard-C-Bibliothek: I/O, Strings, mathematische Funktionen...
  - Weitere Bibliotheken verfügbar (Netzwerk, Grakik, Datenbanken)
  - Eigene Bibliotheken möglich

C++

In den 1960er und 1970er Jahren wurden Programme immer komplexer, schwieriger zu debuggen und oft konnten Projekte nicht im vorgesehenen Zeitrahmen fertiggestellt werden.
Man entwickelte Methoden, große Projekte in kleinere Teilaufgaben aufzuteilen, die einzeln gelöst werden können, und deren Lösungen wiederverwendbar sind. Eine dieser Methoden ist objektorientierte Programmierung.
C++ kombiniert der Geschwindigkeit von C mit modernen (objektorientierten) Programmierkonzepten.

Veränderungen am Sprachkern

- Objektorientierte Programmierung und Klassen → benutzerdefinierte Datentypen
- Funktionen mit gleichem Namen und verschiedenen Argumenten (Überladen)
- Benutzerdefinierte (überladene) Operatoren
- Fehlerbehandlung mit Exceptions

C++-Standardbibliothek

- Ein-/Ausgabeströme (Streams)
- Speicherverwaltung
- Container-Klassen: Listen, Vektoren

C ist fast vollständig als Untermenge in C++ enthalten: ein C++-Compiler kann fast alle C-Programme übersetzen.

ROOT

Mitte der 1990er Jahre stellte sich heraus, dass vorhandene Software (PAW, FORTRAN) nicht für die Analyse der am LHC erwarteten Datenmengen geeignet ist. Als Ersatz wurde auf der Basis von C++ ROOT entwickelt.

Anforderungen an ROOT waren:

- hohe Ausführungsgeschwindigkeit für rechenintensive Aufgaben (Simulation, Rekonstruktion)
- interaktive Shell zur Datenanalyse und -darstellung
- Verwaltung großer Datenmengen: komprimierte Speicherung, schneller Zugriff
- grafische Darstellung, Histogramme etc. wie bei PAW

CINT ist ein C++-Interpreter, mit dem man C++-Programme Schritt für Schritt ausführen und interaktiv arbeiten kann. CINT erlaubt Zugriff auf alle Funktionenn von ROOT.

ROOT I/O verwaltet große Datenmengen in .root-Dateien. Diese Dateien enthalten beliebige Daten, auf die schnell zugegriffen werden kann, und die transparent komprimiert werden.

Histogramme und Grafik erlauben die Darstellung der Ergebnisse direkt aus der Analysesoftware heraus.

Teil II

C und C++

Kapitel 4

Grundlagen in C

4.1 Hello, World!

```
#include <iostream>

using namespace std;

int main()
{

    // Ausgabe eines Grusses
    cout << "Hello, World!" << endl;

    return 0;
}
```

Listing 1: Das erste C++-Programm

Wenn das Listing 1 als `hello.cc` abgespeichert wurde, kann man es so kompilieren und ausführen:

```
$ g++ -o hello hello.cc
$ ./hello
Hello, World!
```

Dateiendungen Für C++-Programme sind die Endungen .cc, .C, .cxx und .cpp üblich. In diesem Skript enden eigenständige Programme in .cc und ROOT-Skripte (auch kompilierbare) in .C.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Abbildung 4.1: Liste aller Schlüsselwörter in C

4.2 Elemente der Sprache C

4.2.1 Schlüsselwörter

Schlüsselwörter bilden den unveränderlichen Kern der Programmiersprache. Im Programmtext dies sind Zeichenfolgen, die in C bzw. C++ eine fest definierte Bedeutung haben.

4.2.2 Bezeichner und Namen

Benutzerdefinierte Elemente wie Variablen und Funktionen besitzen einen Namen, unter dem sie angesprochen. Gültige Namen, auch Bezeichner genannt, dürfen nur aus Groß- und Kleinbuchstaben, Ziffern und dem Unterstrich bestehen und dürfen nicht mit einer Ziffer anfangen.

4.2.3 Operatoren

Operatoren sind in der Regel Sonderzeichen (`+`, `*`, `&`...). Sie symbolisieren Operationen mit Werten wie Addition, Zuweisung, Vergleich etc. In C ist ihre Bedeutung festgelegt.

4.2.4 Literale

Mit Literalen kann ein fester Wert angeben werden. Typische Literale sind Zahlen oder Zeichenketten.

Ganze Zahlen	42	dezimal
	-1	negativ
	0x1F	hexadezimal
	07	oktal
Gleitkommazahlen	1.23	
	2.	
	1.0E2	
Buchstaben	'a'	Buchstabe *a*
	'\n'	Zeilenumbruch
Zeichenketten	"text"	

4.2.5 Kommentare

In C ist alles, was zwischen `/*` und `*/` steht, ein Kommentar und wird ignoriert. C++ erlaubt außerdem `//` als Kommentarzeichen: alles zwischen diesem Zeichen und dem Zeilenende wird ignoriert.

4.3 Einfache Ein-/Ausgabe

Um einfache Ein- und Ausgabe zu ermöglichen, wird hier kurz die Verwendung des C++-Ein-/Ausgabesystems dargestellt. Eine ausführliche Erklärung und Möglichkeiten zur Feineinstellung der Ausgabe werden später erläutert.

4.3.1 Ausgabe mit `cout`

Die Ausgabe im Textfenster erfolgt über `cout`. Es knnen Literale, Variablen oder die Rückgabewerte von Funktionen ausgegeben werden. Als letztes Element sollte immer ein `endl` ausgegeben werden: diese erzeugt einen Zeilenumbruch und veranlasst die eigentliche Ausgabe.

```
cout << "Der Inhalt der Variablen x ist " << x << endl;
```

4.3.2 Eingabe mit `cin`

Mit `cin` kann man einen Wert von der Tastatur in eine Variable einlesen.

```
float x;
cin >> x;
```

	Min	Max	i386	amd64	stdint.h	ROOT
8 bit signed	-128	127	`signed char`	`signed char`	`int8_t`	`Char_t`
8 bit unsigned	0	255	`unsigned char`	`unsigned char`	`uint8_t`	`UChar_t`
16 bit signed	-32 768	32 767	`short`	`short`	`int16_t`	`Short_t`
16 bit unsigned	0	65 535	`unsigned short`	`unsigned short`	`uint16_t`	`UShort_t`
32 bit signed	-2 147 483 648	2 147 483 647	`int`,`long`	`int`	`int32_t`	`Short_t`
64 bit signed	$\tilde{}-10^{19}$	$\tilde{}10^{19}$	`long long`	`long`, `long long`	`int64_t`	`Short_t`

Tabelle 4.1: Integer-Datentypen

4.4 Elementare Datentypen

4.4.1 Ganze Zahlen

engl.: integer numbers → int

- grundlegender Datentyp: Zahlen, Maschinenbefehle, Speicheradressen, Buchstaben, Indizes, Zeit...
- implementiert in Binärdarstellung (100111001)
- mit oder ohne Vorzeichen (`signed` oder `unsigned`)
- verschiedene Breiten: 8, 16, 32, 64 Bit (`char`, `short`, `int`, `long`, `long long`)
- Problem: keine exakte Größenangabe vor C99

Integer-Literale Ganze Zahlen können in dezimaler, oktaler oder hexadezimaler Schreibweise angegeben werden. Zahlen in Hexadezimaldarstellung beginnen mit `0x`, in Oktaldarstellung mit `0` und in Dezimaldarstellung mit einer der Ziffern 1–9. In der Hexadezimaldarstellung werden die Buchstaben A–F oder a–f als Ziffern für 10–15 verwendet.
Beispiel: 42 = 052 = 0x2A

stdint.h in C99 definiert Integer-Typen mit genauer Bitzahl: `uint8_t` `int64_t`

ROOT definiert `UChar_t`, `Char_t`...`Long_t` mit festen Längen.

Typ	Größe	Mantisse	Genauigkeit	pos Min	Max
`float`	32 Bit	24 Bit	10^{-7}	$1.17 \cdot 10^{-38}$	$2.85 \cdot 10^{+45}$
`double`	64 Bit	53 Bit	10^{-16}	$2.23 \cdot 10^{-308}$	$8.09 \cdot 10^{+323}$

Tabelle 4.2: Fließkomma-Datentypen

4.4.2 Buchstaben

werden in C als `char` abgespeichert. Die Zeichen 0-127 sind immer gleich und enthalten Buchstaben, Ziffern, Satzzeichen und einige Sonder- und Steuerzeichen. Die Zeichen 128-255 sind je nach Spracheinstellung des Computers unterschiedlich und enthalten z.B. nationale Sonderzeichen.

Einzelne Buchstaben können mit einfachen Anführungszeichen angegeben werden. Der Wert eines solchen Ausdrucks ist dann der ASCII-Code des Buchstabens.

Beispiel: 'A' = 65

`'\0'`	Ende eines Strings
`'\n'`	Zeilenumbruch
`'\t'`	Tabulator

4.4.3 Fließkommazahlen

engl.: floating point numbers → float

Reelle Zahlen können nur näherungsweise dargestellt werden:

$$m \cdot 2^e \quad m, e \in \mathbb{Z} \tag{4.1}$$

Üblich sind Fließkommazahlen in einfacher (`float`) und doppelter (`double`) Genauigkeit, wie in Tabelle 4.2 beschrieben.

Fließkommazahlen können auch einige spezielle Werte annehmen:

- unenendlich (infinity): der Wert ist unendlich, wobei zwischen $+\infty$ und $-\infty$ unterschieden wird
- NaN (not a number): eine Operation ergibt kein sinnvolles Ergebnis, z.B. sqrt(-1.)

Fließkommakonstanten Fließkommazahlen können als Dezimalzahl mit Punkt oder in halblogarithmischer Darstellung angegeben werden. In der halblogarithmischen Darstellung werden Mantisse und Exponent mit einem e oder E getrennt. Eine Dezimalzahl ohne Punkt wird als ganze Zahl interpretiert.

Beispiel: 3.141528 = 314.1528e-2 = 0.3141528E1

4.4.4 Boolesche (logische) Ausdrücke

In C werden Wahrheitswerte mit Integer-Zahlen dargestellt:

$$\begin{aligned} x = 0 &\Rightarrow \text{falsch} \\ x \neq 0 &\Rightarrow \text{wahr} \end{aligned}$$

In C++ gibt es daneben den Datentyp `bool`, der die Werte `true` und `false` annehmen kann.

4.4.5 `void`

Der spezielle Datentyp `void` kann verwendet werden, wenn explizit keine Daten übergeben werden sollen, z.B. in den Argumenten oder dem Rückgabewert von Funktionen.

4.5 Variablen

Variablen sind ein Zwischenspeicher für Werte eines bestimmten Typs. Sie werden angelegt, indem man den gewünschen Typ und den Namen der Variablen angibt. Der Name muss dabei ein gültiger Bezeichner sein.

```
int n;
float x;
```

Variablen können Werte zugewiesen werden:

```
n = 42;
x = 3.1428;
```

In C++ kann man Variablen auch gleichzeitig anlegen und ihnen einen Wert zuweisen:

```
int n = 42;
float x = 3.1428;
```

Wir haben auch schon gesehen, dass man Variablen von der Tastatur einlesen kann:

```
cin >> n;
cin >> x;
```

4.6 Operatoren

Aus ganzen und Fließkommazahlen können mit Operatoren zu komplexeren Ausdrücken verknüpft werden.

- **unär:** ein Argument, z.B. Vorzeichen
- **binär:** zwei Argumente, z.B. Arithmetik, Vergleiche
- **ternär:** drei Argumente

4.6.1 Arithmetik

Operatoren: `+ - * / %`

Addition, Subtraktion, Multiplikation, Division und Modulo.

Das Ergebnis einer Integer-Division ist wieder eine ganze Zahl: z.B. $1/2 \rightarrow 0$.

Der Modulo-Operator ist nur für ganze Zahlen definiert und berechnet den Rest bei einer Division.

4.6.2 Vergleiche

Operatoren: `== != > < >= <=`

4.6.3 Inkrement und Dekrement

Die Operatoren `++` und `--` kürzen die Addition bzw. Subtraktion um 1 ab.

Steht der Operator vor dem Operanden, wird der Operand zuerst um eins erhöht und dann ausgewertet, steht er nach dem Operanden wird zuerst ausgewertet und dann erhöht.

```
int a=1;
int b=1;

cout << a++ << endl; // gibt 1 aus
cout << ++b << endl; // gibt 2 aus

// a und b sind nun 2
```

4.6.4 Bitweise Operatoren

Bitweise Operatoren verändern die Binärdarstellung einer Zahl.

Schiebeoperatoren : `<<` und `>>` verschieben die Bits einer ganzen Zahl um mehrere Stellen nach links bzw. rechts:

$$\begin{aligned} 1(0001b) \ll 3 &\rightarrow 8(1000b) \\ 127(01111111b) \gg 4 &\rightarrow 7(00000111b) \end{aligned}$$

Bitweise Logik berechnet eine Zahl aus der bitweisen logischen Verknüpfung einer oder zweier Operanden. Beim exklusiven Oder (XOR, `^`) ist das Ergebnis 1, wenn genau einer der Operanden 1 ist.

$$\begin{aligned} 42(00101010b)\&60(00111100b) &\rightarrow 40(00101000b) \\ 42(00101010b)OR60(00111100b) &\rightarrow 62(00111110b) \\ 42(00101010b)\tilde{}60(00111100b)3 &\rightarrow 28(00010110b) \\ \tilde{}42(00101010b) &\rightarrow 213(11010101b) \\ 48\&8 &\rightarrow 0 \end{aligned}$$

4.7 Kontrollstrukturen

4.7.1 Blöcke

Blöcke sind Gruppen von Anweisungen, die in `{ }` engeschlossen sind. Blöcke werden oft verwendet, um in den Schleifen oder Verzweigungen mehrere Anweisungen auszuführen.

In C (aber nicht in C++) müssen alle Variablen am Anfang eines Blockes definiert werden.

4.7.2 `if-else`-Bedingung

```
if (Bedingung) {
  Befehle;
} else {
  Andere Befehle;
}
```

Falls die Bedingung wahr ist, werden die Befehle im ersten Block ausgeführt, ansonsten die Befehle im zweiten Block nach dem `else`. Der `else`-Zweig ist optional und kann weggelassen werden.

`a + b`	Addition
`a - b`	Subtraktion
`a * b`	Multiplikation
`a / b`	Division
`a % b`	Modulo, Rest bei Division
`a++`	Postfix Inkrement
`++a`	Prefix Inkrement
`a--`	Postfix Dekrement
`--a`	Prefix Dekrement
`a == b`	Test auf Gleichheit
`a != b`	ungleich
`a > b`	größer
`a >= b`	größer oder gleich
`a < b`	kleiner
`a <= b`	kleiner oder gleich
`!a`	Negation
`a && b`	Logisches UND (AND)
`a || b`	Logisches ODER (OR)
`a ~ b`	Komplement, bitweise Negation
`a & b`	Bitweises UND (AND)
`a | b`	Bitweises ODER (OR)
`a ^ b`	Bitweises exklusives ODER (XOR)
`a << b`	Verschiebung der Bits in `a` um `b` Stellen nach links
`a >> b`	Verschiebung der Bits in `a` um `b` Stellen nach rechts
`a = b`	einfache Zuweisung
`a += b`	Addition und Zuweisung `a=a+b`
`a -= b`	Subtraktion und Zuweisung
`a *= b`	Multiplikation und Zuweisung
`a /= b`	Division und Zuweisung
`a %= b`	Modulo und Zuweisung
`a &= b`	Bitweises UND und Zuweisung
`a |= b`	Bitweises ODER und Zuweisung
`a ^= b`	Bitweises XOR und Zuweisung
`a <<= b`	Linksverschiebung und Zuweisung
`a >>= b`	Rechtsverschiebung und Zuweisung
`a ? b : c`	Ternäre Bedingung: wenn `a`, dann `b`, sonst `c`
`&a`	Zeiger auf Variable `a`
`*a`	Inhalt des Speichers, auf den `a` zeigt
`a.b`	Zugriff auf Element `b` in `struct a`
`a->b`	Zugriff auf Element `b` in dem `struct`, auf den `a` zeigt

Tabelle 4.3: Operatoren in C, siehe auch `http://en.wikipedia.org/wiki/Operators_in_C_and_C\%2B\%2B`

4.7.3 `switch`-Statement

```
switch (Integer-Ausdruck) {
case 0:
  Befehle;
  break;

case 1:
case 2:
  Befehle;
  break;

default:
  Befehle;
  break;
}
```

Verzweigungen auf der Basis eines Integer-Wertes können als `switch`-Statement geschrieben werden. Die Ausführung springt von `switch` zum passenden `case` oder zu `default`, falls kein `case` passt. Die Ausführung läuft dann weiter bis zum nächsten `break` oder dem Ende des Blocks. Ein erneutes `case` bricht die Ausführung nicht ab.

4.7.4 `while`-Schleife

```
while (Bedingung) {
  Befehle;
}
```

Die Schleife wird ausgeführt, solange die Bedingung wahr ist. Die Überprufung findet vor dem Schleifendurchlauf statt, d.h. wenn die Bedingung von Anfang an falsch ist, wird die Schleife nicht durchlaufen.

4.7.5 `do-while`-Schleife

```
do {
  Befehle;
} while (Bedingung);
```

Im Gegensatz zur `while`-Schleife wird bei der `do-while`-Schleife die Bedingung nach dem Schleifendurchlauf geprüft. Die Schleife wird mindestens einmal durchlaufen.

4.7.6 `for`-Schleife

```
for (Initialisierung ; Bedingung ; Inkrement) {
  Befehle;
}
```

Oft muss man eine Schleife initilisieren , am Beginn jedes Durchlaufs eine Bedingung überprüfen und nach jedem Durchlauf den nächsten Durchlauf vorbereiten.Ein typisches Beispiel ist Listing 2 gezeigt.

```
#include <iostream>

using namespace std;

int main()
{

    for (int i=0; i<10; i++) {
        cout << "Loop " << i << endl;
    }

}
```

Listing 2: `for`-Schleife: die Schleifenvariable `i` wird definiert und mit 0 initialisiert dann wird `i` vor jedem Schleifendurchlauf mit 19 verglichen und falls `i10` ist wird der Text "Loop" und der Wert der Schleifenvariablen ausgegeben. Am Ende der Schleife wird `i` um 1 erhöht.

4.7.7 Abbruch

`break;`	`break` bricht die Ausführung einer Schleife oder eines `switch`-Statements sofort ab.

4.7.8 Schleifenfortsetzung

`continue;`	`continue` bricht die Ausführung eines Schleifendurchlaufs ab, führt ggf. die Inkrementierung einer `for`-Schleife durch und fährt dann mit dem nächsten Durchlauf fort.

4.8 Funktionen

Häufig wiederkehrender Code kann mit Funktionen einmalig geschrieben und mehrfach verwendet werden.

```
int my_function (int argument1, float argument2)
{
  Befehle;
  return 42;
}
```

- Jede Funktion gibt Daten eines bestimmten Typs zurück. Sollen keine Daten zurückgegeben werden, kann der spezielle Typ `void` verwendet werden.

- Die Funktion hat eine lokale Kopie der Argumente → die Argumente können nicht verändert werden.

- Variablen, die innerhalb der Funktion deklariert werden, sind nur innerhalb der Funktion sichtbar.

- Die Funktion kann an jeder stelle mit einem `return` abgeschlossen werden. Nach dem `return` muss der Rückgabewert mit dem passenden Typ angegeben werden.

Beispiel: Summe zweier Zahlen

```
float sum(float a, float b)
{
  float s;

  s = a + b;

  return s;
}
```

Beispiel: Aufruf mit Arrays

Man kann auch Arrays als Funktionsargumente übergeben, muss dann allerdings auch explizit die Größe angeben.

```
float sum(float a[], int n)
{
    float s;

    for (int i=0; i<n; i++) {
        s += a[i];
    }

    return s;
}
```

4.8.1 Deklaration und Implementation

Bevor eine Funktion verwendet werden kann, muss sie deklariert werden. Manchmal möchte man sie an dieser Stelle aber noch nicht implementieren, z.B. weil man alle Funktionen übersichtlich am Anfang der Datei deklarieren möchte.

In diesem Fall kann man eine Forwärts-Deklaration verwenden, bei der statt des Funktionsrumpfes ein Semikolon steht. Die Implementation muss dann später nachgeholt werden.

```
int my_function (int argument1, float argument2);

// weiterer Code
//
// ...

int my_function (int argument1, float argument2)
{
  Befehle;
  return 42;
}
```

Die Deklaration kann auch in einer separaten Datei stehen, die dann in mehrere andere Dateien eingebunden und so wiederverwendet werden kann. Dieser Mechanismus wird oft verwendet, um die in einer Bibliothek definierten Funktionen verwenden zu können.

Solche Dateien werden Header-Dateien genannt und können mit `#include` eingebunden werden:

```
#include "file.h"
#include <string.h>
```

Eckige Klammern werden dabei eher für Header-Dateien des Betriebssystems oder zusätlzlicher Bibliotheken, und Anführungszeichen eher für eigene Header-Dateien verwendet. Es gibt genaue Unterscheidungen, welche Verzeichnisse in beiden Fällen durchsucht werden, aber praktisch kommt man mit dieser Faustregel sehr gut zurecht.

4.8.2 Sichtbarkeit von Variablen — Scope

Variablen sind nur im aktuellen und allen darin eingebetten Blöcken sichtbar. Insbesondere sind Variablen nicht in übergeordneten Blöcken oder anderen Funktionen sichtbar.

Ausserdem gibt es globale Variablen, die überall sichtbar sind. Globale Variablen werden ausserhalb von `main()` und allen anderen Funktionen definiert. Bei globalen Variablen besteht aber die Gefahr den Überblick zu verlieren, welche Teile eines Programms auf diese Variable zugreifen, so dass es vorkommen kann, das eine Änderung an einem Teil des Programm das Verhalten anderer Teile unbeabsichtigt Verändern kann.

4.9 Bibliotheken

Bibliotheken sind Sammlungen von Funktionen, die von mehreren Programmen verwendet werden können. Bibliotheken bestehen aus:

- den **Header-Dateien**, in denen die einzelnen Funktionen beschrieben sind,
- dem übersetzten Maschinencode in einer **dynamischen** oder **statischen Bibliothek** und
- oftmals Manpages (Hilfeseiten), die mit dem Befehl `man` aufgerufen werden können.

Dynamische Bibliotheken werden von den Programmen erst zur Laufzeit geladen und müssen deshalb nur einmal auf der Festplatte gespeichert (installiert) sein. Der Name von dynamischen Bibliotheken endet normalerweise in `.so` (*shared object*, *nix), `.dll` (*dynamically linked library*, Windows) oder `.dylib` (Mac OS X). Statische Bibliothen werden in das ausführbare Programm eingebunden und müssen deshalb nicht separat installiert werden. Ihr Name endet meist in `.a` (*archive*).

4.9.1 Verwendung von Bibliotheken

Um eine Funktion aus einer Bibliothek verwenden zu können, müssen die Header-Dateien mit `#include` in den Programmcode eingebunden werden. Falls die Header-Datei nicht in einem Standardverzeichnis liegt, kann mit der Kompileroption `-I` ein Pfad angegeben werden, in dem nach Header-Dateien gesucht werden soll:

```
g++ -I /path/to/include/files/ program.cc
```

Ausserdem muss das Programm in der Regel mit der Option `-l` gegen die Bibliothek gelinkt werden. Als Argument wird der Name der Bibliothek ohne das einleitende `lib` und die Dateiendung übergeben. Falls die Bibliothek in keinem Stardardverzeichnis liegt, kann man den Suchpfad des Kompilers mit der Option `-L` anpassen.

```
g++ -L /path/to/my/lib -lmylib program.cc
```

4.9.2 `libc` — C-Standardbibliothek

Die C-Standardbibliothek enthält viele Funktionen zur Ein-Ausgabe, Speicherverwaltung, Signalverarbeitung... Da die C-Standardbibliothek in praktisch jedem Programm verwendet wird, wird automatisch gegen sie gelinkt.

Mathematische Funktionen

Mathematische Funktionen sind in C nicht im Sprachkern implementiert, sondern als Teil der C-Standardbibliothek. Um diese Funktionen zu verwenden, muss man in jedem fall die Header-Datei `math.h` einbinden und auf einigen System beim Kompilieren die Option `-lm` angeben.

Die Dokumentation dieser Bibliothek ist unter Unix mit dem Befehl `man` (für Manual) erreichbar. So kann man mit `man math.h` oder `man math` eine Liste aller implementierten Funktionen erhalten. Weitere Informationen zu einzelnen Funktionen erhält man in der Regel durch Aufruf der Man-Page für diese Funktion, z.B. wie in Listing 3 mit `man sin`.

4.10 Felder — Arrays

Felder oder Arrays sind Blöcke von Daten gleichen Typs. Um ein Array anzulegen, muss der Typ und die Anzahl der zu speichernden Elemente angegeben werden:

```
float x[10];
```

Die einzelnen Elemente können durch Angabe des Index in eckigen Klammern angesprochen werden. C nummeriert Indizes ab 0, d.h. die einzelnen Elemente werden im Beispiel mit `x[0]`, `x[1]` ...`x[9]` angesprochen.

Die Größe eines Arrays wird bei der Kompilation festgelegt und kann danach nicht mehr verändert werden. Man muss also schon beim Schreiben des Programms wissen, wieviel Platz der Benutzer später (höchstens) braucht.

Ein Ansprechen von Werten ausserhalb des gültigen Bereichs muss in jedem Fall verhindert werden, da C hier keine eingebauten Schutzmechanismen hat.

C++ kann Arrays bei der Definition auch sofort initialisieren. Dazu werden die gewünschten Werte bei der Definition in geschweiften Klammern angegeben. Die Angabe der Feldgröße kann in diesem Fall weggelassen werden.

```
float a[3] = {2., 3., 4.};
float b[] = {1., 4., 9., 16., 25.};
```

```
SIN(3)                  BSD Library Functions Manual                  SIN(3)

NAME
     sin -- sine function

SYNOPSIS
     #include <math.h>

     double
     sin(double x);

     long double
     sinl(long double x);

     float
     sinf(float x);

DESCRIPTION
     The sin() function computes the sine of x (measured in radians).

SPECIAL VALUES
     sin(+-0) returns +-0.

     sin(+-infinity) returns a NaN and raises the "invalid" floating-point
     exception.

SEE ALSO
     acos(3), asin(3), atan(3), atan2(3), cos(3), cosh(3), sinh(3), tan(3),
     tanh(3), math(3)

STANDARDS
     The sin() function conforms to ISO/IEC 9899:1999(E).

BSD                           December 11, 2006                           BSD
```

Listing 3: Manpage der `sin`-Funktion

4.11 Zeiger

Zu jedem Typ T gibt es einen Zeigertyp "Zeiger auf T", geschrieben `T*`. Ein Zeiger (engl. Pointer) ist die Adresse einer Variablen oder Konstanten im Speicher.

Durch Voranstellen eines `&` erhält man einen Zeiger auf eine Variable.

Die umgekehrte Operation—das Dereferenzieren—kann durch Voranstellen eines `*` erreicht werden.

Ausserdem sind Addition, Subtraktion, Inkrement und Dekrement definiert, wobei eine Veränderung um 1 den Zeiger um die Größe des Objektes im Speicher bewegt.

Zeiger werden verwendet:

- um große Datenmengen an eine Funktion zu übergeben: da eine Funktion immer eine Kopie ihrer Argumente erhält, wäre es ineffizient, große Datenblöcke zu übergeben. Stattdessen übergibt man nur einen Zeiger mit 4 oder 8 Byte.
- um Argumente von Funktionen verändern zu können: wenn man einen Zeiger auf eine Variable übergibt, kann man darüber den Inhalt der Variablen verändern
- zur dynamischen Speicherverwaltung: siehe Kapitel 4.17

Der Zeiger mit dem Wert 0 wird als Null- oder Leerzeiger (engl. Null-Pointer) bezeichnet. Er kann z.B. verwendet werden, um einen Fehler oder einen ungültigen Zeiger anzuzeigen. Ein Dereferenzieren des Nullzeigers ist nicht möglich und bricht die Programmausführung sofort ab.

Der Name eines Arrays (z.B. `int x[5]`) ist auch ein Zeiger auf das erste Element. Die folgenden Ausdrücke sind deshalb gleichwertig:

```
x[i]  ↔  *(x+i)
   x  ↔  &(x[0])
```

4.12 Zeichenketten

In C wird Text (Zeichenketten, engl. Strings) als Array von Buchstaben (`char[]`) verwaltet. Der Text wird mit einer 0 (`'\0'`) abgeschlossen.

```
char mytext[20];
```

- die maximale Länge des Strings wird zur Compile-Zeit festgelegt
- `mytext` ist ein Zeiger auf Buchstaben (`char*`)
- Zuweisungen, Vergleiche etc. funktionieren nicht wie erwartet
- Stringoperation benötigen spezielle Funktionen (`strcmp`, `strcpy`...)

Wegen dieser Nachteile gibt es in C++ den Datentyp `string`, der ein wesentlich intuitiveres Arbeiten mit Zeichenketten erlaubt.

4.12.1 String-Funktionen

Die C-Standardbibliothek bietet Funktionen an, um mit Zeichenketten zu arbeiten. Die Funktionen sind in der Header-Datei `string.h` definiert. Eine Liste dieser Funktionen erhält man in der Manpage `string.h` bzw. `string`.

`size_t strlen(const char *s)` berechnet die Länge einer Zeichenkette bis zum ersten `'\0'`.

`int strcmp(char *s1, char *s2)` vergleicht die beiden Zeichenketten `s1` und `s2`. Das Ergebnis ist 0, wenn beide Strings gleich sind oder -1(+1), wenn der Vergleich `s1` vor(nach) `s2` einordnet.

`int strncmp(char *s1, char *s2, size_t n)` vergleicht die ersten `n` Zeichen der beiden Strings.

`strcoll(const char *s1, const char *s2)` vergleicht zwei Zeichenketten unter Berücksichtigung der Spracheinstellungen.

`int strcpy(char *dest, char *src)` kopiert die Zeichenkette `src` nach `dest`. `dest` muss auf einen ausreichend großen Speicherbereich zeigen.

`int strncpy(char *dest, char *src, size_t n))` kopiert die ersten `n` Zeichen.

`int atoi(char *in)` wandelt eine Zeichenkette in eine ganze Zahl um (definiert in `stdlib.h`).

```
#include <iostream>
#include <iomanip>

#include <string.h>

using namespace std;


int main()
{

    char text1[20] = "Text";
    char text2[20] = "Text";


    if (text1 == text2) {
        cout << "According to ==, '" << text1 << "' and '" << text2
             << "' are equal" << endl;
    } else {
        cout << "According to ==, '" << text1 << "' and '" << text2
             << "' are NOT equal" << endl;
    }



    if (strcmp(text1,text2) == 0) {
        cout << "According to strcmp, '" << text1 << "' and '" << text2
             << "' are equal" << endl;
    } else {
        cout << "According to strcmp, '" << text1 << "' and '" << text2
             << "' are NOT equal" << endl;
    }


}
```

Listing 4.1: C-Strings

`int atof(char *in)` wandelt eine Zeichenkette in eine Gleitkommazahl um (definiert in `stdlib.h`).

4.13 Formatierte Ausgabe in C

Die Ausgabe per `cout` wurde erst mit C++ eingeführt. In C wird Formatierung und Ausgabe über eine Gruppe von Funktionen abgewickelt, die in der Header-Datei `stdio.h` definiert sind.

Die wichtigsten Vertreter sind:

```
int printf(char * format, ...);
int sprintf(char * str, char * format, ...);
int snprintf(char * str, size_t size, char * format, ...);
```

Die Funktion `printf` gibt den formatierten Text auf der Konsole aus, die Funktionen `sprintf` und `snprintf` schreiben die Ausgabe in eine Zeichenkette (die als erstes Argument übergeben wird), wobei man bei `snprintf` noch angeben kann, wie lang die formatierte Ausgabe maximal sein darf. Des weiteren gibt es Funktionen, die in Dateien schreiben können.

Alle diese Funktionen er erwarten einen Format-String und beliebig viele weitere Argumente beliebigen Typs. Die Argumente werden gemäß dem Formatstring formatiert und ausgegeben.

Der Formatstring enthält normalen Text und sogenannte Deskriptoren. Diese Deskriptoren beginnen mit einem `%` und enden mit einem Formatbuchstaben wie in Tabelle 4.4 beschrieben. Zwischen `%` und Formatbuchstabe sind weitere Angaben zur Formatierung möglich:

- **int** gibt die Breite des Ausgabefeldes an
- **.int** gibt die Zahl der Nachkommastellen einer Fließkommazahl an
- **- (Minuszeichen)** gibt die Zahl links- statt rechsbündig aus

`printf` gibt die Anzahl der ausgegebenen Zeichen zurück.

Eine vollständige Beschreibung der Funktionen ist auf der Man-Page zu printf zu finden (`man 3 printf`).

4.14 Kommandozeilenoptionen

Man kann C-Programmen Optionen übergeben, wenn man die `main`-Funktion mit 2 Argumenten definiert:

Buchstabe	Argumenttyp	Umwandlung
d,i	int	Integer-Zahl in Dezimaldarstellung
o	int	Oktalzahl
x,X	int	Hexadezimalzahl
u	int	Ganze Zahl ohne Vorzeichen
c	int	Zeichen (im ASCII-Code)
s	char*	Zeichenkette bis zum abschliessenden `'\0'`
f	double	Fließkommazahl
e,E	double	Fließkommazahl in halblogarithmischer Darstellung
g,G	double	automatische Umstellung zwischen f und e/E
%		Ausgabe des %-Zeichens

Tabelle 4.4: Auswahl von Formatbuchstaben für `printf`

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
  for (int i=0; i<argc; i++) {
    cout << i << ": " << argv[i] << endl;
  }

}
```

Listing 4.2: Kommandozeilenoptionen

```
int main(int argc, char* argv[]);
```

`argc` ist die Anzahl der übergebenen Argumente + 1.

`argv` ist ein Array von C-Strings. Das erste Argument ist der Name des Programms, die weiteren Elemente sind die übergebenen Argumente.

Auf diese Weise können beliebig viele Argumente übergeben werden.

4.15 Strukturen

Der Datentyp `struct` erlaubt, zusammengehörige Daten zusammenzufassen.

```
  struct complex {
```

```
        float re;
        float im;
};

struct complex x;
```

Auf die einzelnen Elemente eines `struct` greift man mit `.` zu.

```
x.re = 0;
x.im = 1;
```

Wenn man einen Zeiger auf einen `struct` hat, kann man das Dereferenzieren des Zeigers und den Zugriff auf ein Element mit `->` zusammenfassen.

```
struct complex *p = &x;
cout << "x=" << p->re << "+" << p->im << "i" << endl;
```

Strukturen werden verwendet:

- wenn Daten geordnet gespeichert werden sollen
- wenn zusammengehörende Daten an eine Funktion übergeben werden
- wenn eine Funktion komplexe Daten zurückgeben soll

4.15.1 Kurzschreibweisen

C++ erlaubt, beim Anlegen von Strukturen das Schlüsselwort `struct` wegzulassen:

```
complex y;
```

4.16 Typumwandlungen

Daten verschiedenen Typs können teilweise ineinander umgewandelt werden. Dazu wird der Typ, den ein Ausdruck annehmen soll, in Klammern vor den Ausdruck gestellt.

```
float x = (float)2;

int int_ptr[2] = {1,2};
long long* long_ptr = (long long*)int_ptr;
```

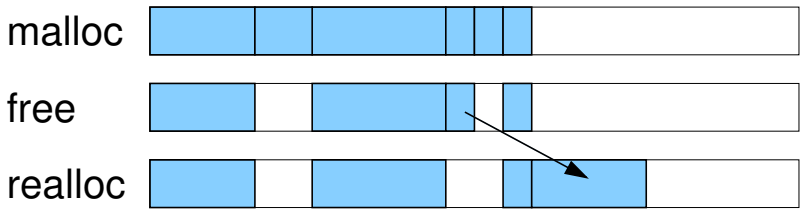


Abbildung 4.2: Heap

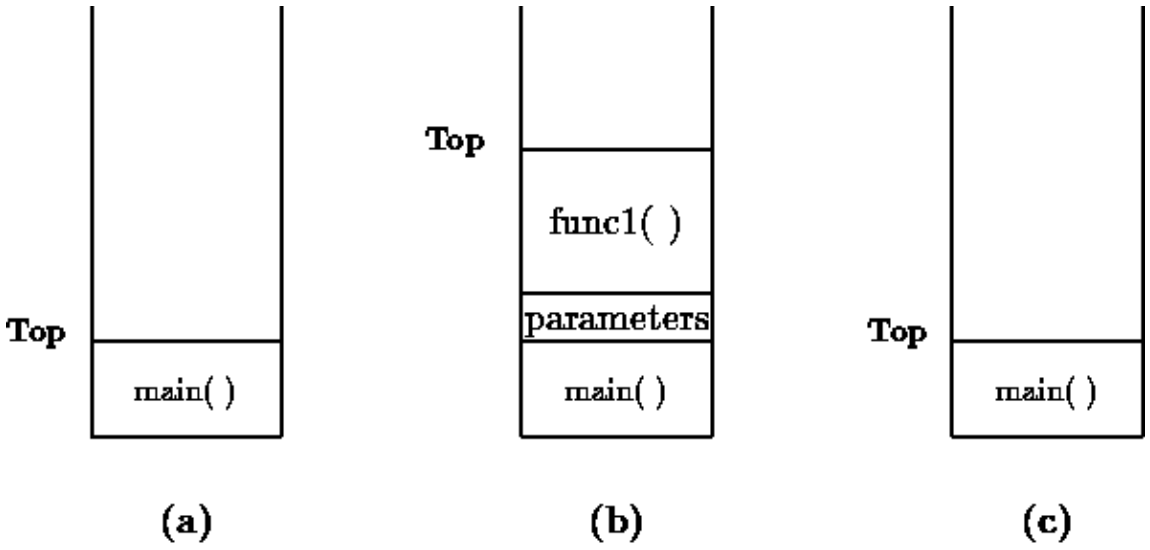


Figure 14.13: Organization of the Stack

Abbildung 4.3: Aufbau des Stack

4.17 Speicherverwaltung

Der Speicher für Variablen oder Arrays wird schon zur Übersetzungszeit reserviert. Zur Laufzeit kann die Anzahl der Elemente in einem Array nicht mehr verändert werden. C bietet die Möglichkeit, zur Laufzeit Speicher anzulegen und wieder freizugeben. Diese Art der Speicherverwaltung heißt “dynamisch”.

Der dynamisch verwaltete Speicherbereich wird **Heap** genannt. Speicher kann in beliebiger Reihenfolge belegt und muss manuell wieder freigeben werden.

Auf dem **Stack** werden alle Parameter und lokalen Variablen von Funktionen angelegt. Die Daten werden auf dem “oberen” Ende des Stacks abgelegt. Wird die Funktion beendet, werden die Daten wieder freigegeben und der Stack schrumpft.

4.17.1 Dynamische Speicherverwaltung mit malloc/free (C)

`void* malloc(size_t size)` fordert einen Speicherbereich mit `size` Bytes an. Diesen Speicherbereich kann man nach entsprechender Typumwandlung als Array benutzen.

```
float* data = (float*)malloc(1000 * sizeof(float));
```

Falls ein Fehler auftritt, gibt `malloc` einen Nullzeiger zurück.

`void* calloc(size_t nmemb, size_t size)` fordert wie `malloc`einen Speicherbereich an, wobei die Gesamtgröße als Zahl und Größe der Elemente angegeben wird.

`void* realloc(void* ptr, size_t size)` vergrößert oder verkleinert einen Speicherbereich unter Beibehaltung des Inhalts. Falls nötig wird der Speicherbereich umkopiert. Der Rückgabewert ist die neue Adresse des Speichers.

`void free(void* ptr)` gibt einen zuvor angeforderten Speicherbereich wieder frei.

Die Speicherverwaltung mit diesen Funktionen hat einige Nachteile:

- der Programmierer muss selbst die Größe des Speicherbereichs berechnen
- da `malloc` einen Zeiger auf void (`void*`) zurückgibt, ist einen explizite Typumwandlung notwendig

4.17.2 Dynamische Speicherverwaltung mit new/delete (C++)

Wegen der genannten Nachteile von malloc/free wurde in C++ die dynamische Speicherverwaltung in den Sprachkern integriert. Ein einzelnes Element (z.B. Variable, Struktur) kann mit `new` angelegt und mit `delete` wieder freigeben werden:

```
struct mystruct {...};
mystruct* x = new mystruct;
delete x;
```

Es gibt auch eine Variante von new/delete, mit der ganze Arrays angelegt werden können:

```
struct mystruct {...};
int n = 5;
mystruct* y = new mystruct[n];
delete[] y;
```

Wichtig: die verschiedenen Methoden zur Speicherverwaltung dürfen nicht gemischt werden, d.h.:

- mit `malloc` allozierter Speicher muss mit `free` freigegeben werden
- mit `new` allozierter Speicher muss mit `delete` freigegeben werden
- mit `new[]` allozierter Speicher muss mit `delete[]` freigegeben werden

Innerhalb eines Programms können alle Methoden verwendet werden, aber es ist zu beachten, dass der Speicher immer mit der passenden Methode wieder freigegeben wird.

4.18 Schritte beim Kompilieren

Der `gcc`-Kompiler unterscheidet vier Schritte beim Kompilieren:

- Präprozessieren
- das eigentliche Kompilieren
- Assemblieren
- Linken

4.18.1 Präprozessor

Der Präprozessor bereitet den Quelltext für die Übersetzung vor. Er bindet include-Dateien ein, führt einfache Textersetzungen aus (sog. Präprozessor-Makros) und kann abhängig von den Makros Codeteile ein- oder ausschließen (Listing 4).

Eine wichtige Präpozessor-Option ist `-I`, mit der Suchpfade für Header-Dateien angegeben werden können.

```
#ifndef HEADER_FILE_HH
#define HEADER_FILE_HH

#endif
```

Listing 4: C-Präprozessor: Schutz vor mehrmaligem Einbinden von Header-Dateien

4.18.2 Kompilieren

Das eigentliche Kompilieren ist das übersetzen des C- oder C++-Quellcodes in Assembler, eine menschenlesbare Darstellung von Maschinencode. Die meisten Kompiler führen in diesem Schritt auch Optimierungen des Codes durch. Dies ist die Hauptaufgabe des Kompilers, alle anderen Schritte sind nur Vor- und Nachbereitung.

Nach dem Präpozessieren bekommt der Kompiler nur genau eine Quellcode-Datei, in der bereits alle Header-Dateien eingefügt wurden. Wenn man einen Kompiler-Fehler vermutet (ein Programmierfehler ist wahrscheinlicher!), kann man diese Datei mit dem Bug-Report einschicken, um z.B. von installierten Header-Dateien unabhängig zu sein.

Da die Ausgabe des Kompilers menschenlesbarer Assembler ist, können erfahrene Programmierer die Ergebnisse nochmals überprüfen.

4.18.3 Assembler

Der Assembler übersetzt den menschenlesbaren Assembler-Code in Nullen und einsen. Das Ergebnis ist eine Objekt-Datei (`.o`), die zwar schon ausführbaren Code enthält, aber nicht direkt ausgeführt werden kann: es fehlen meist noch Funktionen aus Bibliotheken.

4.18.4 Linker

Der Linker verbindet (*to link*) eine oder mehrere Objektdateien und Bibliotheken und erstellt daraus ein ausführbares Programm. Dabei wird überprüft, ob alle verwendeten Symbole (Funktionen, globale Variablen) genau einmal definiert wurden - sonst wird ein Fehler ausgegeben.

Wichtige Linker-Optionen sind `-L` (Suchpfad für Bibliotheken) und `-l` (einzubindende Bibliotheken).

4.19 Inkrementelles Kompilieren

Wurde der Quellcode auf mehrere Dateien verteilt, kann man beim Kompilieren Zeit sparen, indem man nur Dateien neu übersetzt, die auch verändert wurden. Danach muss das Programm allerdings immer auch neu gelinkt werden:

```
g++ -c -o file1.o file1.cc
g++ -c -o file2.o file2.cc
g++ -c -o file3.o file3.cc
```

```
g++ -o prog file1.o file2.o file3.o
edit file1.cc
g++ -c -o file1.o file1.cc
g++ -o prog file1.o file2.o file3.o
```

4.20 Build-Automatisierung mit `make`

Um inkrementelles Kompilieren einfacher handhaben zu können gibt es sogeneannte Build-Systeme, die Abhängigkeiten auflösen und bei Bedarf die nötigen Kommandos in der richtigen Reihenfolge ausführen. Das in der Unix-Welt am weitesten verbreitete Build-System ist `make`.

`make` wird über Makefiles gesteuert, in denen Regeln für alle zu bauenden Objekte stehen. In der Voreinstellung liest `make` die Datei `Makefile` ein. Eine Regel beschreibt einen Schritt, um das Programm zu bauen und unter welchen Bedingungen dies zu bauen ist: was, wann, wie soll gebaut werden:

was? Ziel (Target): Name einer zu erstellenden Datei.

wann? Voraussetzungen (Prerequistes): Dateien, von denen das Target abhängt. Hat sich eine Voraussetzung geändert (d.h. die Datei ist neuer als das Target), wird das Target neu gebaut.

wie? Rezept (Recipe): eine Liste von Befehlen, um das Target neu zu bauen.

```
target: prerequisite ...
<tab>recipe
```

`make` kann mit oder ohne Argumente aufgerufen werden. Wurden Argumente angegeben, so werden diese als Targets interpretiert, die gebaut werden sollen. Wurde kein Argument angegeben, wird das erste Target gebaut, das im Makefile definiert wurde.

4.20.1 Variablen

In Makefiles können Variablen verwendet werden um die Makefiles zu vereinfachen. Variablen sind immer Zeichenketten. Variablen müssen nicht definiert werden, eine Zuweisung reicht aus:

```
VAR=the value can contain spaces
```

Um eine Variable zu verwenden, muss der Variablenname in `$( )` eingeschlossen werden:

```
OTHERVAR=$(VAR)
```

4.20.2 Musterregeln — Pattern Rules

Musterregeln erlauben, ähnliche Regeln nur einmal zu schreiben und für mehrere Targets zu verwenden. Enthält ein Target ein %-Zeichen, so ist dies ein Platzhalter für eine beliebige Zeichenkette. Soll ein Target gebaut werden, auf das dieses Muster passt, werden alle %-Zeichen in den Voraussetzungen ersetzt, und falls diese Voraussetzungen existieren oder gebaut werden können, wird auch das Target gebaut.

Im Rezept können folgende Variablen verwendet werden, um die Regel an den Einzelfall anzupassen:

- `$@`: ist das Target
- `$<`: ist die erste Voraussetzung
- `$^`: sind alle Voraussetzungen

```
%.o: %.c
	gcc -c -o $@ $<
```

Einige Musterregeln sind in `make` bereits vordefiniert, wobei Variablen verwendet werden, mit denen das Verhalten einfach angepasst werden kann:

```
%.o: %.c
	$(CC) $(CFLAGS) -c -o $@ $<

%.o: %.c
	$(CXX) $(CXXFLAGS) -c -o $@ $<

%: %.o
	$(LD) $(LDFLAGS) -o $@ $^
```

4.20.3 clean-Target

Es ist üblich ein Target `clean` zu definieren, mit dem alle bereits gebauten Targets gelöscht werden.

4.20.4 Einschränkungen

`make` kann Abhängigkeiten zwischen Targets nicht selbst bestimmen - diese müssen im Makefile als Voraussetzungen angegeben werden. Es gibt allerdings andere Programme, die entweder als Aufsatz auf `make` konzipiert wurden oder komplett eigenständig laufen, die diese Abhaängigkeiten verwalten können. Beispiele sind die autotools (autoconf+automake), cmake, cons...

Kapitel 5

Objektorientierte Programmierung mit C++

5.1 Paradigmen der Objektorientierung

Abstraktion: "Dinge" in der realen Welt werden durch abstrakte Konzepte dargestellt → abstrakte Datentypen (ADT)

Datenkapselung: auf einen ADT kann nur über die abstrakte Schnittstelle zugegriffen werden - die konkrete Implementation wird vor dem Benutzer versteckt

Vererbung: neue ADTs können durch Abwandlung von bekannten ADTs definiert werden

Polymorphie: (Vielgestaltigkeit) verwandte ADTs können synonym verwendet werden

5.2 Objekt und Klasse

Eine **Klasse** definiert in C++ einen abstrakten Datentyp: sie beschreibt die zu speichernden Daten und die erlaubten Operationen mit diesen Daten. Ein Beispiel wäre das Konzept der komplexen Zahlen (`class complex`).

Ein **Objekt** eine Ausprägung oder Instanz einer Klasse. Das Objekt enthält einen Satz der zu speichernden Daten. Die in der Klassendefinitionen erlaubten Funktionen können auf dieses Objekt angewendet werden.

In C++ wird eine Klasse als Datentyp `class` angelegt, der eine Erweiterung von `struct` ist.

- eine Klasse kann Daten (Attribute) und Funktionen (Methoden) enthalten
- eine Klasse besitzt einen öffentlichen Teil (das Interface), der mit `public:` eingeleitet wird. Auf diesen Teil kann von überall aus zugegriffen werden.
- eine Klasse besitzt einen geschützten und privaten Teil, der mit `protected:` bzw. `private:` eingeleitet wird, und auf den nur die Methoden der Klasse Zugriff haben
- die Methoden können auf die Attribute wie auf lokale Variablen zugreifen

```
class person
{
public:
  void set_name(char* x);
  void set_age(int x);

  char* get_name();
  int   get_age();

private:
  char *name;
  int age;

};
```

Der vollstaändige Name einer Methode besteht aus dem Namen der Klasse, einem '::' und dem Namen der Methode. Dieser vollständige Name muss angegeben werden, wenn eine Methode ausserhalb der Klassendefinition implementiert werden soll.

```
void person::set_age(int x)
{
  age = x;
}

int person::get_age()
{
  return age;
}
```

Der Aufruf einer Methode erfolgt über die Operatoren `.` und `->`, wie beim Zugriff auf die Elemente eines `struct`.

```
person a_person;
a_person.set_name("Max Mustermann");
a_person.set_age(33);

person_t *p = &a_person;

cout << p->get_name() << endl;
```

5.3 Vererbung

Vererbung erlaubt es, neue Klassen durch Erweiterung bereits existierender Klassen zu definieren. Bestehende Attribute und Methoden werden übernommen und können mit neuen Attributen und Methoden ergänzt werden.

```
class student : public person
{
public:
  void set_matriculation_number(int x);
  int get_matriculation_number();

private:
  int matriculation_number;

};
```

Die bereits bestehende Klasse heißt Basisklasse (engl. base class), die neue abegeleitete (engl. derived) Klasse. Im Beispiel ist `person` die Basis- und `student`) die abegeleitete Klasse.

Ein Zeiger auf die abgeleitete Klasse kann wie ein Zeiger auf die Basisklasse verwendet werden. Es stehen dann allerdings nur die Funktionen der Basisklasse zur Verfügung.

```
student a_student;
a_student.set_name("Peter Schmidt");
a_student.set_age(23);
a_student.set_matriculation_number(1234);

person* p = &a_student;
```

```
cout << p->get_name() << endl;

cout << p->get_matriculation_number() << endl; // Fehler!
// person hat keine Methode get_matriculation_number()
```

5.3.1 Zugriffsrechte

Die Methoden einer abgeleiteten Klasse haben Zugriff auf den öffentlichen (`public`) und geschützten (`protected`) Bereich der Basisklasse.

- **public** erlaubt vollen Zugriff auf dieses Attribut oder diese Methode
- **protected** erlaubt nur der Klasse selbst und abgeleiteten Klassen Zugriff
- **private** erlaubt nur der Klasse selbst Zugriff

Die Zugriffsrechte werden in der Klassendefinition mit dem entsprechenden Recht und einem ':' angegeben und gelten bis zur nächsten Angabe. Wird keine Angabe gemacht, ist nur privater Zugriff erlaubt.
Bei der Vererbung können Zugriffsrechte weiter eingeschränkt werden. Oft wird öffentlich vererbt, so dass die abgeleitete Klasse Zugriff auf öffentliche und geschützte Elemente der Basisklasse hat:

```
class derived : public base {...};
```

Bei geschützter und privater Vererbung werden die Rechte bei der Vererbung weiter eingeschrankt. Diese Vererbungsarten sind viel seltener als öffentliche Vererbung.

5.4 Polymorphie

Eine abgeleitete Klasse kann nicht nur die Basisklasse erweitern, sondern auch bestehende Methoden ersetzen. Es wird dann immer die "passende" Version dieser Methode aufgerufen, egal ob über die Basis- oder die abgeleitete Klasse darauf zugegriffen wird.
Die Methode, die ersetzt werden soll, muss als virtuell definiert sein:

```
class person
{

  // ...
```

```
public:
  virtual void print()
  { cout << get_name() << " is a person" << endl; }

  // ...

};

class student
{

  // ...

public:
  virtual void print()
  { cout << get_name() << " is a student" << endl; }

  // ...

};
```

Ein Objekt weiß immer, als welche Klasse es erzeugt wurde, und verwendet dann die entsprechende Methode.

```
person* people[2];

people[0] = &a_person;
people[1] = &a_student;

people[0]->print(); // ruft person::print() auf
people[1]->print(); // ruft student::print() auf
```

5.5 Konstruktor und Destruktor

Eine Klasse kann eine Initialisierungsfunktion bereitstellen, die bei der Erzeugung eines Objekts dieser Klasse aufgerufen wird. Diese Initialisierungsfunktion heißt **Konstruktor**.

- der Konstruktur hat denselben Namen wie die Klasse

- der Konstruktor wird beim Erzeugen eines Objekts als Variable auf dem Stack oder mit `new` auf dem Heap aufgerufen
- ein Konstruktor kann Argumente haben, muss aber nicht
- eine Klasse kann mehrere Konstruktoren mit unterschiedlichen Argumenten haben
- ein Konstruktor hat keinen Rückgabewert (nicht einmal `void`)
- wenn kein anderer Konstruktor definiert ist, wird ein Konstruktor ohne Argumente erzeugt
- ist kein Konstrukor ohne Argumente vorhanden, müssen bei der Erzeugung eines Objekts immer Argumente angegeben werden

Typische Aufgaben des Konstruktors sind:

- sinnvolles Setzen der Attributwerte (z.B. Zeiger auf NULL setzen)
- Anfordern von Speicher (`malloc`)
- Öffnen von Dateien

Ein Konstruktor kann einfach aus einer Funktion bestehen:

```
person::person(char* _name, int _age)
{
  set_age(_age);
  set_name(_name);
}
```

Man kann beim Konstruktor auch eine Initialisierungsliste angeben, mit der Basisklassen und Attribute initialisert werden können.

- ein Attribut eines elementaren Datentyps kann mit dem Attributnamen und dem gewünschten Wert in Klammern initialisiert werden
- ein Attribut, das selbst wieder ein Objekt ist, kann durch Angabe der Argumente für den Konstruktor initialisiert werden
- eine Basisklasse kann durch Angabe ihres Namens und der Argumente eines Konstruktors initialisiert werden

```
student::student(char* _name, int _age, int _matr)
    : person(_name, _age),
      matriculation_number(_matr)
{
}
```

Analog zum Konstruktor gibt es einen **Destruktor**, dessen Name der Klassenname mit vorangestellter Tilde ist. Häufigste Aufgaben des Destruktors sind die Freigabe von Speicher oder das Schließen von Dateien.

5.6 Beispiel: `string`

Da die Verwaltung von Zeichenketten in C problematisch ist, wird in C++ ein neuer Typ `string` eingeführt.

```
string str;

str.assign("Hello, ");
str.append("World!");

cout << str << " has " << str.length() << " characters" << endl;
```

- Speicherverwaltung ist in `string`-Klasse implementiert
- einfachere Benutzung (keine Speicherverwaltung)
- Methoden für Zuweisung, Suchen, Ersetzen, Umwandlung in C-String...
- `http://www.cplusplus.com/reference/string/string/`

C++-Strings sind nicht Bestandteil des C++-Sprachkerns, sondern werden als Klasse in der C++-Standardbibliothek implementiert.

Frage: Wie ist bessere Integration in C++ möglich?

Ziel: Operatoren wie `=`, `+` zum Zuweisen oder Verketten von Strings

5.7 Benutzerdefinierte Operatoren

Um benuzterdefinierte Klassen oder Datentypen (wie z.B. `string`) möglichst nahtlos zu integrieren, erlaubt C++ Operatoren umzudefinieren. Die Operatoren werden

dafür als Funktionen oder Methoden interpretiert, die den Namen `operator` gefolgt von dem zu definierenden Operator tragen.

Wird ein Operator als eigenständige Funktion implementiert, so hat diese Funktion bei unären Operatoren ein, bei binären zwei Argumente.

Wird der Operator als Methode einer Klasse implementiert, so hat ein unärer Operator kein, ein binärer Operator ein Argument. Das linke Argument ist dabei immer das Objekt selbst.

Ein Zuweisungsoperator fuer die Klasse `string` könnte als Methode so implementiert sein:

```
class string
{
  ...

  string operator=(string other)
  {
    assign(other);
    return *this;
  }
}
```

Diese Methode erlaubt es, den Inhalt eines Strings einem anderen String zuzuweisen, (fast) als wären dies interne Datentypen von C++:

```
string s1 = "Hello, World!";
string s2;

s2 = s1;
```

Ein Operator zum Verketten zweier Strings mit einem + könnte als normale Funktion implementiert werden:

```
// ausserhalb der Klasse string
string operator+(string first, string second)
{
  string result = first;
  result.append(second);
  return result;
}
```

Eine Verkettung von Strings kann dann sehr einfach geschrieben werden:

```
string s1 = "Hello, ";
```

```
string s2 = "world!";

s1 = s1 + s2;
```

Entsprechend kann natürlich auch der Operator `+=` neu definiert werden.

5.8 Referenzen

Bei benutzerdefinierten Operatoren möchte man oft die Argumente per Call-by-Reference übergeben. In C muss man hierfür mit dem `&`-Operator einen Zeiger auf die zu übergebende Variable erzeugen, was allerdings nicht dem gewünschen Ergebnis entspricht, dass ich die Klasse wie einen internen Typ verwenden kann.
C++ führt deshalb einen zweiten Mechanismus ein, der Call-by-Reference erlaubt: Referenzen.

- Referenzen sind wie ein Aliasname für ein Objekt
- jede Referenz muss beim Anlegen initialisiert werden
- die Referenz zeigt dann auf das zur Initialisierung verwendetete Objekt, diese Zuordnung kann nicht mehr geändert werden
- Änderungen der Referenz, z.B. Zuweisungen, verändern das referenzierte Objekt

Wie Zeiger beziehen sich auch Referenzen immer auf einen bestimmten Datentyp. Man spricht von einer "Referenz auf T", was in C++ als `T&` geschrieben wird.
Ein etwas nutzloses Beispiel ist:

```
int a=1;
int &b = a; // b ist eine Referenz auf a
b++;

// nun i ist a = b = 2
```

Nützlicher sind Referenzen in Argumenten von Funktionen. Eine Vertauschungsfunktion kann zum Beispiel so geschrieben werden:

```
void swap(int& a, int& b)
{
  int tmp = a;
```

```
    a=b;
    b=tmp;
}

int a=1; int b=2;

swap(a,b);

// nun ist a=2 und b=1
```

In einer Klasse kann man so auch Zugriff auf eigentlich geschützte Attribute gewähren:

```
class myarray
{
  public:
  int &operator[](int x) { return values[x]; }

  protected:
  int values[100];
}

myarray m;
m[0] = 1;

cout << m[0] << endl;
```

5.9 Ein-/Ausgabe-Streams

Ein- und Ausgabe sind die Schnittstelle zwischen den Daten eines Programms und dem Benutzer. Bei der Ausgabe müssen binäre Daten in ein Format umgewandelt werden, das von Menschen gelesen werden kann, bei der Eingabe werden menschenlesbare Daten in binäre Datenstrukturen übersetzt.

C++ ist so konzipiert, dass möglichst nahtlos neue Datentypen (Klassen) integriert werden können. `printf` aus dem C kann allerdings nicht erweitert werden, so dass die neuen Datentypen hier nicht integriert werden konnten. Deshalb wurde für C++ ein neues Ein-/Ausgabesystem entwickelt: Ströme oder Streams.

Die Basisklasse für Ströme ist `ios`, das einen generischen Datenstrom darstellt, der für Ein- oder Ausgabe verwendet werden kann. Davon abgeleitet sind spezielle Klassen, die nur Ein- bzw. Ausgabe beherrschen.

5.9.1 Ausgabeströme

Alle Ausgabeströme erben von der Klasse `ostream`. Die wichtigsten Instanzen dieser Klasse sind `cout` für die Standardausgabe und `cerr` für die Standard-Fehlerausgabe.

Diese Ausgabeströme sind in der Header-Datei `iostream` definiert.

Für die Ausgabe wird der Operator `<<` überladen, der normalerweise als Bitschiebe-Operator bei ganzen Zahlen verwendet wird.

```
cout << 42 << endl;
```

Auf diese Weise können alle im C++-Sprachkern definierten Datentypen ausgegeben werden.

5.9.2 Ausgabemanipulatoren

Oft möchte man die Darstellung der Ausgabe beeinflussen. C++-IO-Streams erlauben dies über Manipulatoren, die wie normale, auszugebende Objekte über den `<<`-Operator an einen Stream übergeben werden und dessen Verhalten verändern.

Die meisten Manipulatoren sind in der Datei `iomanip` definiert.

So kann man mit den Manipulatoren `dec`, `hex` und `oct` zwischen dezimaler, hexadezimaler und oktaler Ausgabe von ganzen Zahlen umschalten:

```
cout << dec << 42 << endl;
cout << hex << 66 << endl;
cout << oct << 26 << endl;
```

5.9.3 Eingabeströme

Alle Eingabeströme erben von der Klasse istream. Wichtigster Vertreter ist die Standardeingabe `cin`, die Daten von der Konsole liest. Der Operator zum Lesen von Daten aus einem Eingeabestrom ist `>>`.

```
float x;
cin >> x;
```

Da bei der Dateneingabe Fehler auftreten können, sollte der Status eines Eingabestroms immer mit folgenden Methoden überprüft werden:

- `eof()`: Dateiende erreicht
- `fail()`: Fehler beim Lesen, z.B. unpassender Datentyp
- `bad()`: IO-Fehler, z.B. Datei nicht lesbar, Datenträger voll

Manipulator	**Bedeutung**
`dec`	Dezimaldarstellung
`hex`	Hexadezimaldarstellung
`oct`	Oktaldarstellung
`setbase(`*n*`)`	Basis n (8,10 oder 16)
`showbase`	`0` vor oktal, `0x` vor hexadezimal
`showpos`	`+` vor positiven Zahlen
`uppercase`	große Buchstaben in hexadezimaler oder halblogarithmischer Darstellung
`scientific`	halblogarithmische Darstellung (1.23e45)
`fixed`	ohne Exponent
`showpoint`	Zeige Dezimalpunkt auch ohne Nachkommastellen
`boolalpha`	Wahrheitswerte als true/false
`left`	linksbündig
`right`	rechtsbündig
`internal`	Leerraum zwischen Vorzeichen und Zahl
`setw(`*n*`)`	Breite des nächsten Ausgabefeldes setzen
`setprecision(`*n*`)`	Ausgabegenauigkeit
`setfill(c)`	Füllzeichen
`flush`	Ausgabepuffer leeren
`endl`	Zeilenende schreiben und Ausgabepuffer leeren

Tabelle 5.1: Ausgabemanipulatoren

- `good()`: keines der oben genannten Probleme

Wird ein Stream in einem boolschen Kontext ausgewertet (z.B. in einer `if`-Bedingung), wird der Status des Streams ausgegeben. Das kann verwendet werden, von einem Stream zu lesen bis der erste Fehler auftritt:

```
float x;
float sum=0.;

while(cin >> x) {
    sum += x;
}
```

5.9.4 Datei-Ströme

Oft sollen Daten in eine Datei geschrieben oder aus einer Datei gelesen werden. Hierfür gibt es die Klassen `ifstream` und `ofstream`, die in der Header-Datei `fstream` definiert sind.

Im Gegensatz zu `cin`, `cout` und `cerr` sind `ifstream` und `ofstream` keine Objekte, sondern Klassen. Das heißt sie müssen vor Benutzung instanziert werden. Dabei kann der Namer der zu öffnenden Datei als Argument des Konstruktors angegeben werden, oder die Datei kann nach der Instanzierung des Stromes mit `open()` geöffnet werden:

```
ifstream if("infile.txt");
ofstream of;

of.open("outfile.txt");

string x;

if >> x;

of << x;
```

Da `ifstream` und `ofstream` von `istream` bzw, `ostream` erben, können alle Operationen und Manipulatoren verwendet werden.

Die Datei kann mit `close()` geschlossen werden, normalerweise kann man das aber dem Destruktor überlassen.

5.9.5 String-Streams

Im C++-Streamsystem gibt es auch ein Analogon zu `sprintfsscanf`: String-Streams. Bei diesen Strömen erfolgt die Ein-/Ausgabe nicht auf der Konsole oder in eine Datei, sondern in einen C++-String innerhalb der Klasse.

Die Klassen für String-Streams heißen `istringstream` und `ostringstream` und sind in der Datei `sstream` definiert.

Der Zugriff auf den String erfolgt über die Methode `str()`:

```
ostringstream os;

os << "Hello, World!" << endl;
cout << os.str() << endl;

istringstream os;

is.str("42 3.1415");

int i;
float x;

is >> i >> x;

cout "1st: " << i << "  2nd: " << x << endl;
```

5.9.6 Benutzerdefinierte Ein- und Ausgabeoperatoren

Eine der mächtigsten Eigenschaften des Streamsystems ist die Möglichkeit, benuzterdefinierte Datentypen in diese System zu integrieren. Dazu muss der Operator `<<` bzw. `>>` überladen werden.

Für die Klasse `person` kann ein Ausgabeoperator folgendermassen definiert werden:

```
ostream& operator<<(ostream& os, person& p)
{
    os << p.get_name() << " (" << p.get_age() << ")";

    return os;
}
```

Ein Objekt des Datentyps `person` kann dann sehr einfach ausgegeben werden:

```
person max("Max Mustermann", 42);
cout << max << endl;
```

Für benutzerdefinierte Eingabe wird entsprechend der Operator >> überladen.

```
istream& operator>>(istream& is, person& p)
{
    string fn, ln;
    int a;

    is >> fn >> ln  >> a ;

    p.set_first_name(fn);
    p.set_last_name(ln);
    p.set_age(a);

    return is;
}
```

Für einfaches Datenlesen reicht dies Möglichkeit aus. Für komplexere Leseaufgaben kann es sich lohnen, einen Parser-Generator zu verwenden (z.B. Boost Spirit).

Der Benuzter kann auch andere Teile des Streamsystems erweitern. So können neue Manipulatoren definiert werden, um die Ausgabe genauer den Erfordernissen anzupassen, oder es können neue Typen von Strömen definiert werden, um z.B. IO mit anderen Systemen über das Netzwerk implementieren zu können.

5.10 Statische Member

Normalerweise erhält jedes Objekt eine Instanz aller Attribute eines Objekts und jede Methode wird implizit mit einem Zeiger auf das Objekt aufgerufen. Manchmal ist es allerdings sinnvoll, Daten nur einmal für alle Instanzen einer Klasse vorzuhalten den Aufruf einer Methode auch ohne bereits existierendes Objekt zuzulassen.

Statische Member werden mit dem Schlüsselwort `static` deklariert.

5.10.1 Statische Attribute

Der Hauptvorteil statischer Attribute ist, dass sie auch bei vielen Objektinstanzen nur einmal Speicherplatz belegen. Unterscheiden sich die Attributwerte für unterschiedliche Instanzen nicht, kann man so den Speicherbedarf des Programmes reduzieren. Die Verwendung statischer Attribute ist deshalb vor allem bei sehr vielen Instanzen oder grosen Attributen sinnvoll.

Statische Attribute können auch in statischen Methoden verwendet werden.

5.10.2 Statische Methoden

Normalerweise kann eine Methode nur für ein bereits existierendes Objekt aufgerufen werden. Manchmal ist es aber sinnvoll, eine Methode aufzurufen, obwohl keine Instanz dieser Klasse existiert, z.B. um eine Instanz zu erzeugen. Diese Methode hat dann keinen Zugriff auf nicht-statische Attribute oder Methoden der Klasse und ähnelt deshalb eher einer normalen C-Funktion. Da Methoden ohnehin nur einmal Speicher belegen, kann man mit statischen Methoden den Speicherbedarf nicht beeinflussen.

5.10.3 Beispiel: Singleton

Ein Singleton ist eine Klasse, die nur einmal instanziert werden kann. Dies kann sinnvoll sein, um den Zugriff auf eine Resource wie eine Datenbankverbindung innerhalb eines Programms zu bündeln: konkurrierende Zugriffe von zwei Instanzen sind auf diese Weise ausgeschlossen.

Listing 5 zeigt eine typische Implementation einer Singleton-Klasse: der Zugriff auf die Klasse erfolgt über die statische Methode `instance()`: existiert bereits eine Instanz der Klasse, wird diese verwendet, ansonsten wird eine neue erzeugt. Ein Zeiger auf die eventuelle Instanz wird als statisches Attribut verwaltet, so dass die `instance`-Methode darauf Zugriff hat. Der Konstruktor ist als privat deklariert, um eine Instanzierung unter Umgehung des Singleton-Mechanismus zu verhindern.

5.11 Namespaces

Möchte man C++-Erweiterungen aus mehreren Quellen verwenden, besteht die Gefahr, dass diese Erweiterungen gleiche Klassen- oder Funktionsnamen verwenden und deshalb nicht gleichzeitig verwendet werden können.

Dies kann vermieden werden, indem jede Erweiterung in einem eigenen Namensraum oder Namespace definiert wird. Solange die Namespaces eindeutig sind (z.B. durch Firmen-, Instituts- oder Experimentname), können diese Erweiterungen dennoch parallel genutzt werden.

Die Definition eines Namespaces wird mit `namespace` und dem Namen des Namespaces eingeleitet:

```
namespace ikp {
  int myfunc();

  class myclass { ... };
};
```

```
#include <iostream>

using namespace std;

class singleton
{
public:
        singleton* instance();

private:
        singleton() {};
        static singleton* the_instance;
};

singleton* singleton::the_instance = 0;

singleton* singleton::instance()
{
        if (!the_instance) {
                the_instance = new singleton;
        }

        return the_instance;
}

int main()
{
        singleton* s = singleton::instance();

}
```

Listing 5: Singleton-Klasse

Die Inhalte eines Namensraums können dann mit dem vorangestellten Namen des Namensraumes und `::` angesprochen werden:

```
ikp::myfunc();
ikp::myclass x;
```

Wenn ein Namensraum sehr häufig verwendet wird, kann man alle Inhalte dieses Namensraumes verfügbar machen:

```
using namespace ikp;

myfunc();
myclass x;
```

Die C++-Standardbibliothek definiert alle Komponenten im Namensraum `std`. Deshalb beginnen alle Beispielprogramme mit `using namespace std`.

5.12 Befreundete Klassen und Funktionen

Manchmal ist es sinnvoll, nur einzelnen Klassen oder Funktionen Zugriff auf ansonsten private oder geschützte Teile einer Klasse zu gewähren. Dazu muss die Klasse oder Funktion, die Zugriff erhalten soll, in der Klasse, auf die zugegriffen werden soll, als `friend` deklariert sein.

```
class A {

  // classes B and C get access to class A
  friend class B;
  friend C;

  // func gets access to class A
  friend int func(A a);
};
```

Teil III

ROOT

Kapitel 6

Das ROOT-Framework

ROOT bezeichnet sich selbst als Framework für die Hochenergiephysik (HEP): ROOT enthält Komponenten, die typische Aufgaben aus der HEP übernehmen:

- Datenvisualisierung
  - Graphen
  - Histogramme
  - Grafik-Primitiven (2D,3D)
- Datenorganisation
  - Container-Klassen: Arrays, Listen, Maps
  - Trees und Ntupel
- Datenspeicherung
- Vektor- und Matrizenrechnung
- grafische Benutzeroberflächen
- parallele Programmierung (PROOF, Threads)
- ...

Diese Funktionen sind als C++-Klassenbibliotheken implementiert, d.h. als Sammlung von Klassen, die als Erweiterung von C++ geladen werden kann.

Um das Arbeiten mit C++ und ROOT einfacher zu machen, gibt es eine interaktive Kommandozeilen-Schnittstelle zu ROOT. Diese interaktive Schnittstelle wird von CINT zur Verfügung gestellt, einem C++-Interpreter, der unabhängig von ROOT entwickelt wurde.

6.1 Die interaktive Shell

Ist ROOT auf einem Rechner installiert, kann mit dem Befehl `root` das Kommandozeilen-Interface ("die Shell") gestartet werden. Nach einer kurzen Begrüßungsmeldung sieht man die Eingabeaufforderung.

An dieser Eingabeaufforderung kann man (fast) beliebige C++-Befehle eingeben. Wenn man die Eingabe nicht mit einem Semikolon beendet, wird das Ergebnis ausgegeben.

```
tom@dietel:~$ root
  ****************************************
  *                                      *
  *        W E L C O M E  to  R O O T     *
  *                                      *
  *   Version   5.26/00  14 December 2009  *
  *                                      *
  *  You are welcome to visit our Web site  *
  *          http://root.cern.ch          *
  *                                      *
  ****************************************

ROOT 5.26/00 (trunk@31882, Dec 14 2009, 20:18:36 on linux)

CINT/ROOT C/C++ Interpreter version 5.17.00, Dec 21, 2008
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0] float x=3.1415;
root [1] x
(float)3.14149999618530273e+00
root [2] cout << x << endl;
3.1415
root [3] for (int i = 0; i<4; i++) cout << "Hello " << i << endl;
Hello 0
Hello 1
Hello 2
Hello 3
root [4]
```

Ausserdem kann man an diesem Prompt auch einige Befehle eingeben, die direkt von CINT ausgeführt werden:

?	Hilfe
.!	Betriebssystem-Befehl ausführen
.L	Datei (Makro, Bibliothek) laden
.x	Makro ausführen
.q	beenden
.qqq	wirklich beenden
.qqqqq	sofort beenden
.qqqqqqq	abbrechen

Tabelle 6.1: Auswahl an ROOT-Befehlen

6.1.1 Laden von Makros und Bibliotheken

Die einfachste Möglichkeit, ROOT zu erweitern, ist durch das Laden von Makros und Bibliotheken. Im einfachsten Fall lädt man beliebigen C++-Code mit:

```
.L macro.C
```

Mit diesem Befehl werden alle in `macro.C` definierten Funktionen und Klassen in ROOT verfügbar gemacht. Das Makro wird dabei nicht kompiliert, sondern interpretiert. Bei dieser Form sind einige Dinge erlaubt, die in kompiliertem C++ nicht möglich sind, z.B. müssen Variablen nicht definiert sein und ROOT-spezifiesche Header-Dateien müssen nicht eingebunden werden.

Man kann das Makro beim Laden auch kompilieren:

```
.L macro.C+
```

Dabei wird das Makro nur dann neu kompiliert, wenn es sich seit dem letzen Kompilieren verändert hat. Will man eine Neukompilierung erzwingen, hängt man zwei `+` and:

```
.L macro.C++
```

Der Vorteil von kompiliertem Code ist vor allem die höhere Ausführungsgeschwindigkeit. Der Code wird ausserdem direkt beim Laden auf Fehler überpruft.

Hängt man zusätzlich noch ein `g` an, so werden Debug-Symbole erzeugt, mit denen man z.B. den Ort eines Laufzeitfehlers besser bestimmen kann:

```
.L macro.C+g
```

Statt mit `.L` kann man ein Makro auch so laden:

```
gROOT->LoadMacro("macro.C")
```

Diese Befehl kann auch in einem Makro stehen, so dass von einem Makro aus andere nachgeladen werden.

6.1.2 Ausführen von Makros

Man kann ein Makro auch laden und sofort eine Funktion daraus ausführen:

```
.x macro.C
```

Dabei muss die Funktion den selben Namen wie das Makro haben, im Beispiel also `macro()`. Durch Anhängen von `+` oder `++` kann man das Makro vor der Ausführung kompilieren.
Benötigt die Funktion Argumente, kann man diese angeben:

```
.x macro.C(5,"hello")
```

6.1.3 Logon- und Logoff-Skripte

Existiert im aktuellen Verzeichnis ein Skript `rootlogon.C` oder `rootlogoff.C` wird dieses beim Starten bzw. Beenden von ROOT ausgeführt.

6.1.4 Kommandozeilenoptionen von ROOT

Man kann ein Makro direkt beim Aufruf von ROOT ausführen lassen:

```
root macro.C
```

Argumente können ebenfalls angegeben werden, wobei man beachten muss dass die speziellen Zeichen der `bash` (`(`, `)`, `"`) mit Anfüuhrungszeichen oder Backslash maskiert werden:

```
root 'macro.C(5,"hello")'
```

Ausserdem kann man weitere Argumente übergeben:

- **b** Batch-Mode ohne Grafikausgabe
- **l** Begrüßungsgrafik wird nicht angezeigt
- **n** Logon- und Logoff-Skripte werden nicht ausgeführt
- **q** ROOT wird nach ausführen des Makros beendet

Kapitel 7

Objekte in ROOT

7.1 Klassenhierarchie

Die Funktionalität von ROOT ist als Klassenbibliothek implementiert. ROOT ordnet alle Klassen in einer einzigen Klassenhierarchie an, in der alle Klassen—direkt oder indirekt—von `TObject` erben.

7.1.1 `TObject`

`TObject` stellt das grundlegende Interface zur Verfügung, das jede Klasse in ROOT implementieren muss. Über die Methoden von `TObject` wird sichergestellt, dass die Funktionen von ROOT für alle Klassen zur Verfügung stehen.

Grafik: `Draw()` und `Paint()` zeichnen das Objekt. In `TObjekt` tun diese Methoden nichts, sie müssen ggf. überladen werden.

Textausgabe: `Print()` zeigt Informationen über das Objekt an

Ein-/Ausgabe: Objekte können direkt in ROOT- oder XML-Dateien gespeichert werden, oder als Makros, mit denen die Objekte neu erzeuft werden können. Hierfür stehen Funktionen wie `Write()`, `Streamer()`, `SaveAs()` zur Verfügung.

Klonen: ROOT-Klassen können mit der Methode `Clone()` vervielfältigt werden.

Container: einige ROOT-Klassen können weitere Objekte enthalten. Der Zugriff darauf kann z.B. mit `FindObject()` erfolgen.

Run-Time Type Interface: im Gegensatz zu C++ können in ROOT von jedem Objekt zur Laufzeit Typinformationen abgefragt werden.

- `ClassName()` gibt den Namen der Klasse zurück
- `InheritsFrom(const char *classname)` gibt 1 zurück, wenn die Klasse von `classname` erbt, sonst 0

7.1.2 `TNamed`

Viele ROOT-Objekte sollen eindeutig identifizierbar sein. Deshalb gibt es eine Klasse, die `TObject` um einen Namen und einen Titel erweitert: `TNamed`. Der Zugriff auf Name und Titel erfolt mit `GetName()` und `GetTitle()`. Beim Klonen ist als Argument zu `Clone()` der Name des neu erzeugten Objekts anzugeben, ansonsten ist das Objekt identisch.

Kapitel 8

ROOT Grafik

8.1 Einfache Geometrische Objekte

Die Basis für viele komplexere Diagramme in ROOT sind einfache Grafikelemente wie Linien, Text, Boxen, die sogenannten Grafik-Primitiven.

8.1.1 `TLine`

Eine `TLine` zeichnet eine gerade Line, wobei Farbe, Breite und Linienart festgelegt werden können.
Sehen wir uns den Stammbaum von `TLine` an:

```
class TLine : public TObject, public TAttLine { /* ... */ };
```

`TLine` erbt von `TObject`:

- `Draw()` fügt die Linie in eine Grafik ein. Diese Methode wird vom Benutzer aufgerufen, um eine Linie zu zeichnen.
- `Paint()` zeichnet die Linie auf den Bildschirm. Diese Methode wird nicht vom Benutzer, sondern nur von ROOT intern aufgerufen.

und von `TAttLine`

- `SetLineColor()/GetLineColor()`: Farbe setzen und zurücklesen
- `SetLineWidth()/GetLineWidth()`: Linienbreite setzen und zurücklesen
- `SetLineStyle()/GetLineStyle()`: Linienart setzen und zurücklesen

`TAttLine` enthält alle Attribute, die eine Linie in ROOT haben kann, und ist auch die Basis für andere Klassen, die Linien darstellen müssen, wie z.B. `TBox`, `TEllipse`. `TAttLine` garantiert, dass alle diese Klassen das selbe Interface zum setzen von Linienattributen verwenden.
Zur Erzeugung von `TLine` gibt es drei Konstruktoren:

- `TLine()` erzeugt eine Linie von (0,0) nach (0,0) und wird vor allem für Linien verwendet, die nicht direkt gezeichnet werden sollen, sondern z.B. mit `DrawLine()` neue Linien erzeugen.
- `TLine(Double_t x1, Double_t y1,Double_t x2, Double_t y2)` erzeugt eine Linie von (x1,y1) nach (x2,y2)
- `const TLine &line)` ist der Copy-Konstruktor, der verwendet wird, wenn eine `TLine` als Kopie einer anderen `TLine` erzeugt werden soll.

Weitere Methoden sind:

- `DrawLine(Double_t x1, Double_t y1, Double_t x2, Double_t y2)` erzeugt eine Kopie dieser `TLine`, setzt Start- und Endpunkt und fügt die Kopie der aktuellen Grafik hinzu.
- `DrawLineNDC(Double_t x1, Double_t y1, Double_t x2, Double_t y2)`: wie `DrawLine()`, jedoch im NDC-Koordinatensystem

`TLine` enthält die Datenfelder `fX1`, `fY1`, `fX2` und `fY2`, die Start- und Endpunkt der Linie festlegen.

8.1.2 Weitere Grafik-Primitiven

- `Arrow`: Pfeile
- `TPolyLine`: Linien aus mehreren geraden Segmenten, kann auch als Polygon verwendet werden
- `TMarker`: einzelner Markierungspunkt (z.B. Datenpunkt)
- `TPolyMarker`: mehrere Markierungspunkte
- `TBox`: Rechteck
- `TEllipse`: Ellipsen und Kreise
- `TText`: Text
- `TLatex`: Text mit LaTeX-ähnlichen Formatierungen

8.2 Attribute

Neben dem bereits bekannten `TAttLine` gibt es noch Attribute für Flächen, Markierungspunkte und Text.

8.2.1 `TAttLine`

- `SetLineColor()`: Farbe setzen und zurücklesen
  z.B. `kBlack`, `kRed`, `kGreen`, `kBlue`
- `SetLineWidth()`: Linienbreite setzen und zurücklesen, Breite 1.0 ist eine schmale Line
- `SetLineStyle()`: Linienart setzen und zurücklesen, z.B. `kSolid`, `kDashed`, `kDotted`, `kDashDotted`

8.2.2 `TAttFill`

Das Füllen von Flächen wird durch eine Farbe und ein Muster gesteuert:

- `SetFillColor` setzt die Farbe
- `SetFillStyle` setzt das Muster

8.2.3 `TAttText`

- `SetTextAlign` setzt die horizontale (10=links 20=mittig 30=rechts) + vertikale Ausrichtung (1=unten 2=mittig 3=oben)
- `SetTextAngle` setzt den Winkel, mit dem die Schrift dargestellt wird
- `SetTextColor` setzt die Farbe
- `SetTextSize` setzt die Schriftgröße relativ zur Padhöhe

8.2.4 `TAttMarker`

- `SetMarkerColor` setzt die Farbe
- `SetMarkerStyle` setzt die Form des Markierungspunktes, wie sie von `TMarker::DisplayMarkerTyp`
  dargestellt werden
- `SetMarkerSize` setzt die Größe des Punktes in Einheiten von 8 Pixeln

8.3 Farben

Alle Attributklassen verwenden Farben, die in ROOT als ganze Zahlen implementiert sind. Die Anordnung der Farben in der Palette ist willkürlich, so dass man Farben mit einer Tabelle auswählen sollte. Eine solche Tabelle kann mit der Methode `TCanvas::DrawColorTable()` erzeugt werden:

```
cnv = new TCanvas();
cnv->DrawColorTable();
```

Alternativ kann ROOT ein Farbrad darstellen, aus dem die Farben ausgewählt werden können:

```
wheel = new TColorWheel;
wheel->Draw();
```

In diesem Farbrad sind einige Farben mit Namen (kBlue, kRed...) aufgeführt. Ähnliche Farben können durch Addition oder Subtraktion kleiner Zahlen erreicht werden.

8.4 Pad und Canvas

Wenn ROOT Grafikausgabe produziert, öffnet sich ein neues Fenster, in dem diese Ausgabe erscheint. Dieses Fenster wird in ROOT von der Klasse `TCanvas` gesteuert. `TCanvas` ist eine Spezialisierung der Klasse `TPad`.

`TPad` ist ein Bereich auf dem Bildschirm, in den ROOT zeichnen kann. Das `TPad` verwaltet hierfür eine Liste aller Objekte, die in diesem Bereich gezeichnet werden sollen. Wird ein Objekt mit `Draw()` gezeichnet, so wird dieses Objekt einfach dieser Liste hinzugefügt.

`TCanvas` ist ein Pad in einem Fenster, das vom Betriebssystem dargestellt werden kann.

8.4.1 Unterpads

Ein `TPad` kann weitere `TPad`-Instanzen enthalten, die dann innerhalb des Pads gezeichnet werden:

```
subpad1 = new TPad("subpad1","The first subpad",.1,.1,.5,.5);
subpad1->Draw();
```

Für das Teilen eines Pads in ein Raster von Unterpads gibt es eine Abkürzung:

```
pad->Divide(2,3);
```

Dieser Befehl erzeugt 6 Unterpads zu `pad`, die in 2 Spalten und 3 Zeilen angeordnet sind.

Obwohl viele Pads gleichzeitig dargestellt werden können, ist immer nur ein Pad aktiv: in diesem Pad werden alle Objekte gezeichnet. Das aktive Pad ist unter der globalen Variablen `gPad` erreichbar.

8.4.2 Wann wird ein Pad gezeichnet?

Der Inhalt eines Pads auf dem Bildschirm wird nicht bei jedem Zeichnen eines Objekts in das Pad aufgefrischt, da dies zu langsam wäre. Stattdessen wird zu bestimmten Zeitpunkten überprüft, welche Pads sich verändert haben und nur diese Pads werden neu gezeichnet.

Dazu hat jedes `TPad` ein Bit, das anzeigt, ob das Pad verändert wurde. Dieses Bit wird gesetzt

- durch Anklicken des Pads mit der Maus, z.B. durch Vergößern
- beim Beenden eines Scripts
- beim Hinzufügen oder Ändern von Primitiven
- von der Methode `TPad::Modified()`

Die Ausgabe erfolgt erst, wenn das `TCanvas`, das das Pad enthält, aktualisiert wird. Dies passiert, wenn

- ein Befehl in CINT abgearbeitet wurde oder
- die Methode `TCanvas::Update()` aufgerufen wird

8.5 Koordinatensysteme

2D-Grafik in ROOT unterscheidet drei Koordinatensysteme: benuzterdefinierte Koordinaten, NDC-Koordinaten und Pixel-Koordinaten.

8.5.1 Benutzerdefinierte Koordinaten

Am häufigsten werden benutzerdefinierte Koordinaten verwendet: die Lage des Ursprungs und die Skalierung der Achsen kann vom Benutzer (oder einem komplexeren Grafikobjekt) frei gewählt werden. Die Koordinaten beziehen sich dann immer auf die dargestellten Daten, z.B. eines Histogramms.

Die meisten Grafikbefehle erwarten benutzerdefinierte Koordinaten.

Benutzerdefinierte Koordinaten lassen sich mit der Methode `TPad::Range` setzen:

```
pad->Range(0., 0., 30., 200.);
```

Normalerweise setzt ein Histogramm oder Graph die benutzterdefinierten Koordinate automatisch.

8.5.2 Normalisierte Koordinaten

Bei normalisierten oder NDC-Koordinaten liegt der Koordinatenursprung in der linken unteren Ecke des Pads und die Seitenlängen des Pads sind auf 1 normiert. Die rechte obere Ecke hat damit die Koordinaten (1,1).

Mit NDC-Koordinaten lassen sich Grafik-Objekte an einer definierten Stelle innerhalb des Pads statt relativ zu den dargestellten Daten platzieren.

8.5.3 Pixel-Koordinaten

Pixel-Koordinaten werden sehr selten und fast nur von ROOT intern verwendet. Sie zählen die Pixel, ausgehend von der linken oberen Ecke des Fensters.

8.5.4 Lineare und logarithmische Darstellung

Ein Pad kann die Achsen unabhängig voneinander linear oder logarithmisch darstellen. Zum Wechseln dienen die Methoden `TPad::SetLogX()` und `TPad::SetLogY()`: werden die Methoden ohne Argument oder dem Argument 1 aufgerufen, wird die entsprechende Achse logarithmisch dargestellt, nach einem Aufruf mit dem Argument 0 linear. `TPad::SetLogZ()` schaltet die Darstellung der Z-Achse in dreidimensionalen Darstellungen um.

8.6 Text

Die vielfältigste Klasse zur Darstellung von Text ist `TLatex`. `TLatex` ahmt den Mathematikmodus von LaTeX nach. Der Hauptunterschied ist, dass statt des Backslash (“\”) ein Pfund-Symbol (“#”) die LaTeX-Kommandos einleitet.

Kapitel 9

Datenvisualisierung

9.1 Graphen

Die Klasse `TGraph` repräsentiert Datenpunkte mit jeweils 2 Koordinaten. Primär dient die Klasse der Darstellung, man kann aber auch einige Kenngrößen wie Mittelwert und RMS abfragen und Fits durchführen.

9.1.1 Erzeugung

Graphen werden häufig schon bei der Erzeugung mit Datenpunkten gefüllt. Die meisten Konstruktoren erwarten deshalb zwei Arrays oder Vektoren als Argumente.

```
TGraph(Int_t n, const Int_t* x, const Int_t* y)
TGraph(Int_t n, const Float_t* x, const Float_t* y)
TGraph(Int_t n, const Double_t* x, const Double_t* y)
TGraph(const TVectorF& vx, const TVectorF& vy)
TGraph(const TVectorD& vx, const TVectorD& vy)
```

Man kann die Datenpunkte auch direkt aus einer Datei lesen:

```
TGraph(const char* filename,
       const char* format = "%lg %lg",
       Option_t* option = "")
```

Dieser Konstruktor liest die Datenpunkte gemäs `format` aus `filename`. `format` verwendet dabei das selbe Format wie `scanf`, die Voreinstellung `%lg %lg` liest also die ersten beiden Felder jeder Spalte ein.

9.1.2 Darstellung

Die Darstellung verwendet wie üblich die `Draw`-Methode. Die Darstellungsart kann dabei als Option übergeben werden.

- `L` Poly-Linie, die alle Punkte verbindet
- `C` glatte Kurve durch alle Punkte
- `P` ein Marker an jedem Punkt
- `*` ein Stern an jedem Punkt

Normalerweise wird der Graph dem aktuellen Pad hinzugefügt, und es werden keine Achsen gezeichnet. Um den Inhalt des aktuellen Pads zu löschen und den Graph mit Achsen zu zeichnen, gibt man die Option `A` an:

- `A` Padinhalt löschen un Achsen zeichnen

Groß- und Kleinschreibung ist bei den Optionen unwichtig. Es können auch mehrere Optionen gleichzeitig angegeben werden.

`TGraph` erbt von `TAttLine`, `TAttFill` und `TAttMarker`, so dass das Aussehen des Graphen wie üblich verändert werden kann.

9.1.3 Achsen

Die gezeichneten Achsen sehen wahrscheinlich noch nicht wie gewünscht aus: zumindest fehlen noch die Achsenbeschriftungen.

Die Achsen des Graphen sind Instanzen der Klasse `TAxis`. Diese Klasse regelt die Beschriftung der Achsen. `TAxis` erbt von der Klasse `TAttAxis`, die das Aussehen der Klasse festlegt.

Die Achsen werden erst erzeugt, wenn ein Graph schon gezeichnet wurde. Dann kann man mit `GetXaxis()` und `GetYaxis()` auf sie zugreifen:

```
TAxis *xaxis = graph->GetXaxis();
```

Danach kann die Achsenbeschriftung gesetzt werden:

```
xaxis->SetTitle("title (unit)");
```

Weitere Darstellungsoptionen können über die Klasse `TAttAxis` gesetzt werden:

Die **Farbe der Achse und der Teilstriche** kann mit gesetzt werden:

- `SetAxisColor(Color_t color = 1)`

Der **Titel** der Achse zeigt i.d.R. die aufgetragene Größe und deren Maßeinheit an. Farbe, Schriftart, Größe und Abstand zur Achse können verändert werden:

- `SetTitleColor(Color_t color = 1)`

- `SetTitleFont(Style_t font = 62)`
- `SetTitleOffset(Float_t offset = 1)`
- `SetTitleSize(Float_t size = 0.04)`

Labels sind die Zahlenangaben an der Achse. Auch hier können Farbe, Schriftart, Größe und Abstand zur Achse verändert werden:

- `SetLabelColor(Color_t color = 1)`
- `SetLabelFont(Style_t font = 62)`
- `SetLabelOffset(Float_t offset = 0.005)`
- `SetLabelSize(Float_t size = 0.04)`

Die Zahl der **Unterteilungen** kann ebenfalls eingestellt werden. Man unterscheidet 3 Unterteilungsebenen: die gesamte Achse wird normalerweise in maximal 10 primäre Teile geteilt (`n1`), die jeweils in 5 sekundäre Teile geteilt werden (`n2`). Optional steht eine dritte Unterteilungsebene zur Verfügung (`n3`). Es stehen zwei Funktionen, zur Verfügung, um die Unterteilungen zu konfigurieren, die entweder `n1`,`n2` und `n3` oder `n=n1+100*n2+10000*n3` erwarten. Der optionale Parameter steuert, ob die Zahl der Unterteilungen fest verwendet wird, oder ob ROOT versucht, die Darstellung zu optimieren, indem auch weniger als die angegebenen Unterteilungen verwendet werden dürfen.

- `SetNdivisions(Int_t n = 510, Bool_t optim = kTRUE)`
- `SetNdivisions(Int_t n1, Int_t n2, Int_t n3, Bool_t optim=kTRUE)`

Auch läßt sich die Länge der Teilstriche einstellen:

- `SetTickLength(Float_t length = 0.03)`

9.1.4 Graphen mit Fehlerbalken

Da in der Physik immer Fehler anzugeben sind, kann ROOT mit der Klasse `TGraphErrors` auch Graphen mit Fehlerbalken erzeugen.

Es stehen die selben Konstruktoren wie für `TGraph` zur Verfügung, erweitert um zusätzliche Felder für die Fehler in X und Y:

```
TGraphErrors(const TVectorF& vx, const TVectorF& vy,
             const TVectorF& vex, const TVectorF& vey)
TGraphErrors(const TVectorD& vx, const TVectorD& vy,
             const TVectorD& vex, const TVectorD& vey)
TGraphErrors(Int_t n, const Float_t* x, const Float_t* y,
             const Float_t* ex = 0, const Float_t* ey = 0)
TGraphErrors(Int_t n, const Double_t* x, const Double_t* y,
             const Double_t* ex = 0, const Double_t* ey = 0)
TGraphErrors(const char* filename,
             const char* format = "%lg %lg %lg %lg",
             Option_t* option = "")
```

Die Darstellung der Fehler kann als Balken, Rechtecke, Striche oder Band erfolgen. Die Steuerung erfolg über Optionen der `Draw`-Methode:

- `Z` Weglassen der Fehlerstriche
- `X` keine Fehler
- `[]` nur Fehlerstriche, keine Fehlerbalken
- `2` Rechtecke
- `3` Fehlerband

Asymmetrische Fehler sind mit der Klasse `TGraphAsymmErrors` möglich.

9.1.5 Kenngrößen

Graphen erlauben, die wichtigsten Eigenschaften der Verteilungen in X und Y abzufragen:

- `GetMean(1)`: $\mathrm{E}(X)$
- `GetMean(2)`: $\mathrm{E}(Y)$
- `GetRMS(1)`: $\sigma_X = \sqrt{\mathrm{Var}(X)}$
- `GetRMS(2)`: $\sigma_Y = \sqrt{\mathrm{Var}(Y)}$
- `GetCovarianve()`: $\mathrm{Cov}(X, Y)$
- `GetCorrelationFactor()`: $\mathrm{Cov}(X, Y)/\sqrt{\mathrm{Var}(X)\mathrm{Var}(Y)}$

9.2 Histogramme

Histogramme werden in der Kernphysik oft verwendet, um Häufigkeitsverteilungen zu studieren.

- grafische Darstellung von Häufigkeitsverteilungen
- Unterteilung des Wertebereich einer Variablen in mehrere Bins
- Hochzählen des Bin-Inhalts bei jedem Ereignis innerhalb des Bins
- mehrdimensionale Histrogramme für 2,3 (oder mehr) Variablen
- Gesamtzahl an Bins = Produkt der Bins je Dimension
- 1,2,3-dimensionale Histogramme leiten sich in ROOT von `TH1`, `TH2` bzw. `TH3` ab
- der Inhalt jedes Bins kann als 8-bit char (`THxC`), 16-bit short (`THxS`), 32-bit int (`THxI`), float (`THxF`) oder double (`THxD`) gespeichert werden

Die Dimension eines Histogramms ist die Zahl der Variablen, die abgebildet werden soll. Eindimensionale Histogramme leiten sich von der Klasse `TH1` ab, zweidimensionale von `TH2` und dreidimensionale von `TH3`. Dabei ist zu beachten, dass die Gesamtzahl an Bins das Produkt der Bins je Dimension ist.
Beispiel: ein TH3D mit 100 Bins je Achse belegt bereits $100 \times 100 \times 100 \times 8B = 8MB$

9.2.1 Erzeugung von Histogrammen

Der einfachste Konstruktor von `TH1x` erwartet Name und Titel des zu erzeugenden Histogramms sowie die Zahl der zu erzeugenden Bins sowie Minimum und Maximum des Wertebereichs.

```
TH1F h1("h1", "My First Histogram", 40, 0., 10.);
```

Der zweite Konstruktor erlaubt, beliebige Bin-Grenzen zu definieren:

```
Float_t bins[] = {0., 1., 2., 3., 4., 5., 10., 20.};
TH1F h2("h2", "My Second Histogram", 7, bins);
```

Bei der Erzeugung von zwei- und dreidimensionalen Histogrammen müssen die Achsendefinition entsprechend oft angegeben werden.

9.2.2 Füllen

Histogramme werden mit der Methode `Fill()` gefüllt. Bei jedem Aufruf wird der Inhalt des entsprechenden Bins um 1 erhöht.

```
h1D->Fill(1.);
h2D->Fill(1., 42.);
h3D->Fill(1., 42., 666.);
```

Der `Fill`-Methode kann ein Gewicht als optionales weiteres Argument übergeben werden. Der Inhalt des Bins wird dann nicht um 1, sondern um diese Gewicht erhöht.

```
h1D->Fill(1., 0.5);
h2D->Fill(1., 42., 2.0);
h3D->Fill(1., 42., 666., 1.0);
```

9.2.3 Darstellung 1-dimensionaler Histogramme

`Draw()` zeichnet ein Histogramm in das aktive Pad. Bereits vorhandene Grafik in dem Pad wird dabei gelöscht. Diese Methode hat einen optionalen Parameter, der die Art der Darstellung bestimmt:

`B`	Säulendiagramm
`L`	Poly-Linie, die alle Punkte verbindet
`C`	glatte Kurve durch alle Punkte
`P`	ein Marker an jedem Punkt
`*`	ein Stern an jedem Punkt
`E`	Fehlerbalken

9.2.4 Darstellung 2-dimensionaler Histogramme

2-dimensionale Histogramme können entweder in 2D durch Punktwolken, Konturen oder Farben dargestellt werden, oder in 3 Dimensionen als Höhenprofile

`SCAT`	Punktwolke (Voreinstellung)
`BOX`	Rechtecke, Größe proportional zu Bininhalt
`COL`	Farben gemäß einer Palette
`CONT`	Konturen gleicher Werte
`LEGO`	Säulendiagramm
`SURF`	interpoliertes Oberflächendiagramm

Die Darstellungen `CONT` und `SURF` können durch Anhängen von Ziffern weiter modifiziert werden.

9.2.5 Darstellung 3-dimensionaler Histogramme

3-dimensionale Histogramme sind oftmals sehr unübersichtlich und sollten nur selten verwendet werden.

`SCAT`	Punktwolke (Voreinstellung)
`BOX`	Quader, Größe proportional zu Bininhalt
`ISO`	3D-Iso-Fläche durch Mittelwert der Bininhalte

9.2.6 Fehlerbalken

Normalerweise werden Fehler als die Wurzel des Bin-Inhaltes bestimmt:

$$\Delta y_i = \sqrt{y_i}$$

Das Histogramm geht also von einer Poisson-Verteilung der Werte in jedem Bin aus.

Beim Füllen mit Gewichten ist dies allerdings nicht korrekt. Bei einfach gewichteten Einträgen ist der Fehler $\Delta e_i = 1$. Der Fehler wächst mit dem Gewicht, so dass frr-gewichtete Einträge $\Delta e_i = w_i$ gilt. Quadratische Summierung ergibt dann für den Fehler des Bininhaltes:

$$\Delta y_i = \left(\sum_{i=1}^{N} w_i^2 \right)^{1/2}$$

Um die Fehler auf diese Weise zu berechnen, muss vor dem Füllen die Methode `Sumw2()` aufgerufen werden, um die Datenstrukturen zum Berechnen der Quadratsumme der Gewichte anzulegen.

9.2.7 Operationen mit Histogrammen

Mit der `Scale`-Methode können alle Bins eines Histogramms mit einem konstanten Wert multipliziert werden. Dies ermöglicht z.B. das Normieren eines Histogramms auf Binbreiten oder die Umrechnung von Ereignisraten in Wirkungsquerschnitte.

Die Methoden `Add`, `Divide` und `Multiply` addieren, dividieren bzw. multiplizieren zwei Histogramme bin-weise. Dabei können jeweils ein oder zwei Histogramme und zu jedem Histogramm ein Skalierungsfaktor übergeben werden.

Jede dieser Methoden gibt es in 3 Varianten, hier am Beispiel `TH1::Add`:

- `TH1 TH1::Add(const TH1* h1, Double_t c1 = 1)` addiert `h1` zum aktuellen Histogramm
- `TH1 TH1::Add(const TH1* h, const TH1* h2, Double_t c1 = 1, Double_t c2 = 1)` addiert die Histogramme h1 und h2 und speichert das Ergebnis im aktullen Histogramm
- `TH1 TH1::Add(TF1* f, Double_t c1 = 1, Option_t* option = )` addiert die Werte der Funktion `f` zum aktuellen Histogramm. Normalerweise wird der Wert in der Bin-Mitte addiert; falls die Option `Ï"` angegeben wurde, wird das Integral innerhalb des Bins addiert.

Subtraktion kann durch addieren mit einer Skalierung von -1 erreicht werden.
Die Operationen können auch mit den Operatoren `+ - * /` durchgeführt werden.

9.2.8 Projektionen

Für ein gegebenes zweidimensionales Histogramm kann man die Häufigkeitsverteilung für eine der beiden Achsen erhalten, indem man das Histogramm auf diese Achse projiziert:

```
TH1 *px = hist2D->ProjectionX();
```

Als Argumente kann der Name des projizierten Histogramms sowie erster und letzter zu projizierender Bin auf der jeweils anderen Achse angegeben werden:

```
TH1 *px = hist2D->ProjectionX("px", 10, 20);
```

Dieser Befehl projiziert die Y-Bins 10-20 des 2D-Histogramms `hist2d` auf die X-Achse und nennt das neue Histogramm "px".
Analog lassen sich auch 3D-Histogramme auf eine oder zwei Achsen projizieren.

9.2.9 Profile

Wird bei einem 2D-Histogramm nur der Mittelwert und die Breite der Verteilung in Y-Richtung benötigt, kann man auch Profile verwenden. Der Vorteil gegenüber 2D-Histogrammen ist der geringere Speicherbedarf und eine evtl. klarere Darstellung.
Der Konstruktor von `TProfile` erwartet 6 Argumente:

```
TProfile::TProfile(char* name, char* title,
                   int nbins, float xlow, float xhigh,
                   Option_t* opt);
```

Die ersten 5 Argumente entsprechen den Argumenten für ein `TH1D`. Das 6. Argument gibt an, wie die Breite der Verteilung als Fehlerbalken dargestellt werden soll. Andere Konstruktore erlauben, beliebiges Binning einzustellen oder den Wertebereich für Y einzuschränken.
Das `TProfile` berechnet intern für jeden Bin j:

- $H_j = \sum y$
- $E_j = \sum y^2$
- $N_j = \sum 1$

Option	Fehler			Kommentar
	$s_j > 0$	$s_j = 0 \wedge N_j > 0$	$s_j = 0 \wedge N_j = 0$	
`" "`	$s_j/\sqrt{N_j}$	$\sqrt{H_j}/\sqrt{N_j}$	0	Fehler des Mittelwerts (Voreinstellung)
`"s"`	s_j	$\sqrt{H_j}$	0	Breite der Verteilung
`"i"`	$s_j/\sqrt{N_j}$	$1/\sqrt{12N_j}$	0	Fehler des Mittelwerts unter der Annahme ganzzahliger Messwerte

Tabelle 9.1: Optionen zur Berechnung der Verteilungsbreiten bei `TProfile`

Das RMS der Verteilung läßt sich dann einfach ausrechnen:

$$s_j = \sqrt{\frac{E_j}{N_j} - \frac{H_j^2}{N_j^2}}$$

Soll ein `TProfile` mit Gewichten gefüllt werden, muss `Sumw2` aufgerufen werden, die Summen werden dann entsprechend angepasst.

Je nach Option, die dem Konstruktor übergeben wurde, werden die Fehlerbalken wie in Tabelle 9.1 dargestellt.

9.2.10 Allgemeine Darstellungsoptionen

Histogramme erben von `TAttLine`, `TAttFill` und `TAttMarker`, die entsprechende Darstellungsprmen beeinflussen.

Weitere Methoden, mit denen die Darstellung beeinflusst werden kann, sind:

- `SetTitle()` setzt den Titel des Histogramms, der normalerweise in der linken oberen Ecke des Canvas dargestellt wird
- `SetXTitle()` setzt den Titel der X-Achse
- `SetYTitle()` setzt den Titel der Y-Achse
- `SetZTitle()` setzt den Titel der Z-Achse
- `SetStats(Int_t) stat` zeigt eine Statistik-Box mit den wichtigsten Kenngrößen des Histogramms an

Ausserdem gibt es noch weitere Optionen zur `Draw`-Methode:

`SAME`	zeichnet weitere Histogramme in bestehende Grafik
`HIST`	zeichnet Histogramm ohne Fehler und assoziierte Funktionen
`FUNC`	zeichnet nur assoziierte Funktionen

9.3 Pad-Eigenschaften

9.3.1 Lineare und Logarithmische Darstellung

Jede der 2 bzw. 3 Achsen eines Pads kann mit den Funktionen `SetLogx()`, `SetLogy()` und `SetLogz()` zwischen logarithmischer und linearer Darstellung umgeschaltet werden. Ohne Argument oder mit dem Argument 1 wird logarithmische, mit dem Argument 0 lineare Darstellung ausgewählt.

9.3.2 Festlegung benutzerdefinierter Koordinaten

Graphen, Histogramme und Funktionen legen die benutzerdefinierten Koordinaten beim ersten Zeichnen in ein Pad fest. Soweit die anderen Objekte über diesen Bereich hinuasragen, werden sie nicht dargestellt. Um dies zu umgehen, muss man manuell den Bereich vorgeben.

Die kann man erreichen, indem man ein leeres Histogramm erzeugt, das den kompletten zu zeichnenden Bereich abdeckt, und dieses zuerst zeichnet.

```
TH1C frame("frame","", 100, -1., 1.);
frame.SetMinimum(2.);
frame.SetMaximum(3.);
frame.Draw();
```

Zum Zeichnen kann die Achse beliebig grob unterteilt sein, z.B. nur aus einem einzelnen Bin bestehen. Man kann allerdings nur auf ganze Binbreiten in ein solches Histogramm hineinzoomen, so dass es sinnvoll sein kann, hierfür wesentlich mehr Bins anzulegen als zum Zeichnen notwendig.

9.3.3 Pad-Ränder

An den Seiten eines Pads sind Ränder, die für Achsenbeschriftungen, Titel und ähnliches genutzt werden können. Zum Anpassen dieser Ränder stehen vier Methoden von `TPad` bereit:

- `SetTopMargin(Float_t margin)`
- `SetBottomMargin(Float_t margin)`
- `SetLeftMargin(Float_t margin)`
- `SetRightMargin(Float_t margin)`

Die Ränder werden relativ zur Padbreite bzw. -höhe gesetzt.

9.4 Funktionen

Die Klasse `TF1` implementiert mathematische Funktionen mit einem bestimmten Definitionsbereich und kann diese grafisch darstellen. Die Funktion hat ein Argument `x` und beliebig viele freie Parameter.

9.4.1 Erzeugung mit Formeln

Die Funktion kann mit dem Konstruktor als Formel spezifiziert werden, wobei neben dem Argument `x` noch viele Standard-C Funktionen und die statischen Methoden der Klasse `TMath` verwendet werden können:

```
TF1 f1("f1","sin(x)/x",-3,3);
TF1 f2("f2","TMath::Log(x)",0,3);
```

Die freien Parameter können in der Definition der Funktion mit `[0]`, `[1]`... benutzt werden:

```
TF1 f3("f3","[0]*x*sin([1]*x)",-3,3);
```

Bevor eine solche Funktion gezeichnet werden kann, müssen die Parameter noch gesetzt werden:

```
f3.SetParameter(0, 1.2);        // set 1st parameter
f3.SetParameter(1, 3.5);        // set 2nd parameter
f3.SetParameters(1.2, 3.5);     // set 2 parameters at once

Double_t pars[] = {1.2, 3.5};
f3.SetParameters(pars);         // set parameters using array
```

Man kann den Parametern einer Funktion auch Namen geben:

```
f3.SetParName(0,"amplitude");
f3.SetParNames("amplitude", "frequency");

f3.SetParameter("amplitude", 1.2);
f3.SetParameter("frequency", 3.5);
```

9.4.2 Erzeugung mit C-Funktion

Die Definition einer `TF1` kann sich auf eine bereits bekannt C-Funktion beziehen, indem man dem Konstruktor einen Zeiger auf diese Funktion übergibt:

```
Double_t myfunction(Double_t *x, Double_t *par)
{
  Float_t xx =x[0];
  Double_t f = TMath::Abs(par[0]*sin(par[1]*xx)/xx);
  return f;
}

TF1 myfunc("myfunc",myfunction,0,10,2);
```

9.4.3 Erzeugung mit Funktor

Eine Klasse, die den `operator()` implementiert, nennt man in C++ oft ein Funktionsobjekt oder einen Funktor. Man kann einen solchen Funktor als Argument fuer den Konstruktor einer `TF1` verwenden:

```
class MyFunctionObject {
  public:
  double operator() (double *x, double *p) {
    // ...
  }
};

MyFunctionObject * fobj = new MyFunctionObject(....);
TF1 * f = new TF1("f",fobj,0,1,npar,"MyFunctionObject");
```

Der Vorteil einer Funktor-Klasse gegenüber einer einfachen C-Funktion ist, dass die Klasse weitere Variablen und Methoden enthalten kann, soa dass der Funktor beliebiges Verhalten implementieren kann.

9.4.4 Zeichnen von Funktionen

Beim Zeichnen von Funktionen mit der Methode `Draw` kann eine Option angegeben werden:

`L`	gerade Linien zwischen den berechneten Punkten (Voreinstellung)
`C`	glatte Kurve durch alle berechneten Punkte
`FC`	gefüllte Fläche unter einer glatten Kurve
`SAME`	Funktion in bestehende Grafik einzeichnen

Wird die Option `SAME` nicht angegeben, so wird der Inhalt des aktuellen Pads gelöscht, die benutzerdefinierten Koordinaten des Pads werden Definitions- und Wertebereich der Funktion angepasst, und die Achsen sowie die Funktion werden gezeichnet.

Um eine Funktion mehrmals mit verschiedenen Parametern in ein Pad zu zeichnen, müssen mehrere Kopien dieser Funktion angelegt werden. Die Methode `DrawCopy()` erleichtert dies, da das kopieren und zeichnen in einem Schritt erreicht wird.

Die Funktion wird normalerweise für 100 Punkte berechnet. Falls dies nicht ausreicht, kann man den Wert mit der Methode `SetNpx()` verändern.

9.4.5 Mehrdimensionale Funktionen

Neben `TF1` gibt es noch die Funktionen `TF2` und `TF3`, die Funktionen der Typen $\mathbb{R}^2 \to \mathbb{R}$ bzw. $\mathbb{R}^3 \to \mathbb{R}$ implementieren.

9.5 Fits

Graphen und Histogramme können mit einer Funktion gefittet werden, d.h. freie Parameter einer Funktion können so bestimmt werden, dass Funktion und die Datenpunkte des Graphen bzw. Histogramms möglichst gut übereinstimmen.

Normalerwise führt ROOT χ^2-Fits durch, d.h. ROOT sucht den Parametersatz für den diese Größe minimal wird:

$$\chi^2 = \sum_{i=1}^{N} \frac{\left(f(x_i, \vec{p}) - y_i\right)^2}{\sigma_i^2}$$

Neben der bestmöglichen Anpassung der Funktion erhält man aus dem Fit auch informationen darüber, wie gut die Funktion die Daten beschreibt. Man erwartet, dass das reduzierte χ^2 nahe bei der Zahl der Freiheitsgrade liegt, also der Zahl der Datenpunkte abzüglich der Zahl der zu bestimmenden Parameter:

$$\chi^2 \approx N_{dof} = N_{data} - N_{par}$$

9.5.1 Interaktive Fits

9.5.2 Programmatische Fits

Will man Fits nicht immer manuell durchführen, kann man die Methode `Fit` von Graphen und Histogrammen durchführen.

```
TFitResultPtr TH1::Fit(TF1* f1,
                       Option_t* option = "", Option_t* goption = "",
                       Double_t xmin = 0, Double_t xmax = 0);
```

Option	Bedeutung
`Q`	minimale Ausgabe
`V`	maximale Ausgabe
`W`	setze alle Fehler = 1
`L`	Log-Likelihood Fit
`I`	benutze Integral statt Wert in Bin-Mitte
`R`	verwende Definitionsbereich der Funktion
`+`	vorhandene assoziierte Fit-Funktionen werden nicht gelöscht
`S`	Rückgabe der Ergebnisse als `TFitResult`
`N`	Fit wird nicht gezeichnet

Tabelle 9.2: Ausgewählte Optionen fur Fits

```
TFitResultPtr TH1::Fit(char* formula,
                       Option_t* option = "", Option_t* goption = "",
                       Double_t xmin = 0, Double_t xmax = 0);
```

Die gefittete Funktion wird dem Histogramm bzw. Graphen als assoziierte Funktion hinzugefügt. Vorhandene assoziierte Funktione werden dabei gelöscht.

9.5.3 Zugriff auf Ergebnisse

Auf die Ergebnisse des Fits können über die assoziierte Funktion des Graphen bzw. Histogramms zugegriffen werden, die man über die Methode `GetFunction(char* name)` erhält:

```
TF1* fit = hist->GetFunction("gaus");
```

Danach kann man χ^2 und Fitparameter mit Fehlern abfragen:

```
fit->GetChisquare();
fit->GetParameter(0);
fit->GetParError(0);
```

Wird dem Fit die Option `''s''` übergeben, so wird das Fit-Ergebnis nicht nur in der Fit-Funktion abgelegt, sondern auch als `TFitResult` zurückgegeben. Über diese Klasse kann auch auf die Kovarianzmatrix des Fits zugegriffen werden.

```
TFitResultPtr r = h1.Fit(&gaus,"QS","E");
r->Chi2();
r->Parameters()[0];
r->Errors()[0];
r->GetCovarianceMatrix();
```

9.5.4 Fit-Status

Falls ein Fit nicht konvergiert oder sonstige Probleme auftreten, sind die Ergebnisse oft nicht verläßlich. Der Status des Fits wird im `TFitResultPtr` abgespeichert, der Zugriff darauf erfolgt, indem man den `TFitResultPtr` in eine ganze Zahl umwandelt:

```
int result = hist.Fit(...);
```

War der Fit erfolgreich, wird 0 zurückgegeben, ansonsten eine Zahl ¿ 0, die weitere Informationen über die Fehlerquelle enthält.

9.5.5 Startparameter

Die Anfangswerte für die Minimierung können der anzupassenden Funktion mit `SetParameters()` übergeben werden.

9.5.6 Einschränken von Parametern

Man kann einzelne Parameter auf einen beschränkten Bereich einschränken, oder im Fit sogar komplett festhalten.

```
TF1 gaus("gaus","gaus", 0., 10.);
gaus.FixParameter(1, 5.);          // fix mean at 5
gaus.SetParLimits(2, 0.5, 1.5)   // limit 0.5 < sigma < 1.5
```

9.5.7 Log-Likelihood Fits

Der χ^2-Fit hat Probleme, wenn einige der Bins keine Einträge enthalten. In diesem Fall verwendet der χ^2-Fit den Punkt nicht, wobei Information verloren geht.

Für Fits an Histogramme mit sehr wenigen Einträgen und Bins ohne Eintraëge sollten deshalb Log-Likelihood-Fits verwendet werden. In diesem Fit wird für jeden Bin unter der Annahme einer Poisson-Statistik eine bedingte Wahrscheinlichkeit (Likelihood) berechnet, und die Summe dieser bedingten Wahrscheinlichkeiten wird dann maximiert.

Ein Log-Likelihood-Fit wird durchgeführt, wenn die Option `''L''` übergeben wird.

Kapitel 10

ROOT-I/O

- Speichern und Laden von Datenstrukturen: Persistenz
- Kompression zur effizienten Speicherung
- Erweiterbarkeit: alle Klassen, die von `TObject` erben, können gespeichert werden
- Zugriff auf lokale `.root`-Dateien
- Zugriff über das Netzwerk mit `rootd`
- Zugriff über Apache-Webserver per HTTP

10.1 ROOT-Dateien

ROOT legt Daten in Dateien ab, die in ROOT von der Klasse `TFile` repräsentiert werden. In jeder Datei können beliebig viele Objekte gespeichert werden, die alle von `TObject` erben müssen.

As Besonderheit kann eine Datei weitere Unterverzeichnisse enthalten, die wiederum Objekte und weitere Unterverzeichnisse enthalten können. Diese Verzeichnisse werden in der Klasse `TDirectory` implementiert. Da `TFile` von `TDirectory` erbt, steht die gesamte Funktionalität von Verzeichnissen auch für Dateien zur Verfügung.

10.1.1 Öffnen von ROOT-Dateien

ROOT-Dateien werden geöffnet, indem ein Objekt vom Typ `TFile` erzeugt wird.

```
TFile(const char* fname, Option_t* option = "",
      const char* ftitle = "", Int_t compress = 1)
```

- `fname` ist der Name der Datei, die geöffnet werden soll
- `option` gibt an, wie die Datei geöffnet werden soll:
  - `NEW"` oder `CREATE"` legt eine neue Datei an. Existiert diese Datei bereits, schlägt das Öffnen fehl
  - `RECREATE"` legt die Datei neu an oder überschreibt eine existierende Datei
  - `ÜPDATE"` öffnet die Datei zum Schreiben und legt sie ggf. neu an
  - `READ"` öffnet eine Datei zum Lesen (Voreinstellung)
- `ftitle` angezeigter Name in Browsern
- `compress`: 0 (keine Kompression), 1 (geringe Kompression, hohe Geschwindigkeit) ... 9 (hohe Kompression, geringe Geschwindigkeit)

Nach dem Öffnen einer Datei kann man mit `ls()` den Inhalt anzeigen:

```
TFile f("example.root");
f.ls();
```

10.1.2 Navigation im Verzeichnisbaum

Man kann mit `cd()` in eine Datei oder ein Unterverzeichnis wechseln:

```
f.cd();
f.cd("subdir");
```

Die globale Variable `gDirectory` zeigt immer auf aktuelle Verzeichnis. Dies kann man verwenden, um das aktuelle Verzeichnis anzuzeigen:

```
gDirectory->pwd();
gDirectory->ls();
```

CINT kennt für `gDirectory->ls()` die Abkürzung `.ls`.

10.1.3 Lesen von Objekten

Um schnell zu den einzelnen Einträgen in einem Verzeichnis oder einer Datei springen zu können, enthält jedes Verzeichnis eine Liste von `TKey`-Objekten, die jeweils den Namen und einen Verweis auf ein Objekt in der Datei enthalten.

Diese Schlüsselliste kann man sich anzeigen lassen:

```
gDirectory->GetListOfKeys()->Print()
```

Mit den Methoden `Get` und `GetObject` kann man gezielt einzelne Objekte innerhalb eines Verzeichnises anhand ihres Namens ansprechen. Die Methode `Get` gibt das Objekt über einen Zeiger auf `TObject` zurück, der explizit in den ürsprünglichen Typ umgewandelt werden muss.

Diese Typumwandlung, mit der Gefahr von Inkompatibilitäten, lässt sich mit der Methode `GetObject` vermeiden, die vor der Zuweisung eine Typüberprüfung durchführt. Falls der Typ des Objekts in der Datei nicht mit dem Zeigertyp kompatibel ist, wird ein Nullzeiger zurückgegeben.

```
TH1* hist = 0;
gDirectory->Get("hist", hist);
if (hist==0) { /* Error handling */ };
```

10.2 Schreiben von Objekten

Jedes Objekt, das von `TObject` erbt, kann durch Aufruf der Methode `Write` in das aktuelle Verzeichnis geschrieben werden.

```
hist.Write();
```

Als erstes Argument kann der Name angegeben werden, unter dem das Objekt in der Datei gespeichert werden soll. Dies ist sinnvoll, wenn der Name beim Schreiben geändert werden soll, oder wenn das Objekt nicht von `TNamed` erbt.

Wird ein Objekt mehrmals geschrieben, werden mehrere Versionen abgespeichert. Die einzelnen Versionen werden unterschieden, indem an den Namen ein Semikolon und die Zyklusnummer angehängt wird. Zyklusnummern beginnen bei 1 und werden für jede neue Version um 1 hochgezählt.

Wird ein Objekt mit dem Namen “hist” abgespeichert, so wird die erste Version als “hist;1”, die zweite als “hist;2” usw. abespeichert.

Wird als zweites Argument von `Write` `TObject::kOverwrite` oder `TObject::kWriteDelete` angegeben, so wird ein vorhandener Zyklus überschrieben. `TObject::kOverwrite` löscht die alte Version bevor die neue geschrieben wird und ist deshalb schneller aber unsicherer als `TObject::kWriteDelete`.

Da jeder Schreibzugriff langsam ist und jeder Zyklus Speicherplatz benötigt, sollten Objekte so selten wie möglich geschrieben werden.

Ein neues Unterverzeichnis kann mit der Methode `TDirectory::mkdir` erzeugt werden:

```
TDirectory *dir = f.mkdir("dirname", "title of subdirectory");
```

10.3 Löschen von Objekten

Mit der Methode `Delete` können Objekte wieder gelöscht werden.

```
dir->Delete("foo");    // Objekt "foo" im Speicher
dir->Delete("foo;1"); // Objekt "foo", Version 1
dir->Delete("foo;*"); // alle Versionen von Objekt "foo"
dir->Delete("*;1");    // Version 1 aller Objekte
```

10.4 Benutzerdefinierte Klassen

Man kann auch benutzerdefinierte Klassen in `.root`-Dateien ablegen, wenn sie von `TObject` erben. Ausserdem muss jede Klasse, die geschrieben werden soll, eine Methode `Streamer` enthalten, die das Objekt in eine Byte-Folge und wieder zurück umwandeln kann. Diesen Streamer kann man von ROOT erzeugen lassen, indem man in der Definition der Klasse das Macro `ClassDef` mit Klassenname und einer Versionsnummer aufruft und ausserhalb der Definition das Macro `ClassImp` mit dem Klassennamen als Argument. Die Versionsnummer dient zur Unterscheidung inkompatibler Versionen der Klasse.

```
class myclass : public TObject
{
  // ...

  ClassDef(myclass,1);
};

ClassImp(myclass);
```

Diese Klasse kann dann wie alle anderen ROOT-Klassen in Dateien geschrieben und wieder zurückgelesen werden.

Normalerweise werden alle Komponenten einer Klasse geschrieben. Man kann allerdings für einzelne Felder durch bestimmte Kommentare ein besonderes Verhalten festlegen:

- `//!`: die Komponente ist transient, sie wird nicht gespeichert
- `//[n]`: die Komponente ist ein Array eines elementaren Datentyps oder einer Klasse mit `n` Elementen, wobei `n` eine weitere Komponente der zu speichernden Klasse ist
- `//->`: die Komponente ist ein gültiger Zeiger, für den die `Streamer`-Methode aufgerufen werden kann

- `//||`: beim Speichern in einem Tree (siehe nächstes Kapitel) wird diese Komponente nicht gesplittet

Es ist auch möglich, eine benutzerdefinierte statt der von ROOT erzeugten `Streamer`-Methode zu verwenden. Damit können z.B. transiente Komponenten nach dem lesen auf sinnvolle Werte gesetzt werden.

Kapitel 11

Trees

Normales ROOT-I/O ist darauf ausgelegt, relativ wenige Objekte zu speichern, während in vielen Experimenten sehr viele identisch aufgebaute Datensätze zu speichern sind. Diese Ähnlichkeit kann genutzt werden, um Daten effizienter zu speichern. Normalerweise werden Objekte immer zusammenhängend gespeichert. Will man dann bei einer großen Anzahl Objekte jeweils nur eine Komponente abfragen, muss man trotzdem alle Objekte vollständig in den Speicher laden.

Trees

- speichern größe Mengen gleichartiger Objekte ab
- Komponenten der Objekte werden getrennt abgespeichert (Branches)
- nur benötigte Branches müssen gelesen werden
- effiziente Kompression ähnlicher Daten in einem Branch

11.1 Ntupel

Ntupel (`TNtuple`) sind Trees, die bis zu 14 `float`-Werte je Eintrag speichern können. Da dies ein sehr häufiger Einsatzzweck ist, wurde hierfür eine spezielle Klasse definiert.

11.1.1 Ntupel anlegen und füllen

Der Konstruktor von `TNtuple` benötigt neben Name und Titel noch die Namen der Datenfelder, die durch Doppelpunkte getrennt als drittes Argument angegeben werden:

```
TNtuple nt("nt", "the title of the ntuple", "x:y:z");
```

Einträge können dem Ntupel mit der Methode `Fill` hinzugefügt werden:

```
nt.Fill(1.2, 3.4, 5.6);
```

11.1.2 Grafische Darstellung

Die Daten eines Ntupels können in 1, 2 oder 3 Dimensionen dargestellt werden, wobei die zu zeichnenden Feldnamen als erstes Argument, durch Doppelpunkte getrennt angegeben werden:

```
nt.Draw("x");
nt.Draw("x:y");
nt.Draw("x:y:z");
```

Statt einzelner Feldnamen können auch komplexe Ausdrücke mit mehreren Feldnamen und mathematischen Funktionen verwendet werden:

```
nt.Draw("x:y-sin(z)");
```

Als zweites Argument kann eine Formel angeben, die als Gewicht für den jeweiligen Eintrag verwendet wird. Wenn die Funktion nur boolesche Werte (0 und 1, z.B. Ergebnis von Vergleichen) zurückgibt, kann man damit auch Cuts erreichen:

```
nt.Draw("x", "y");    // histogram x with a weight of y
nt.Draw("x", "y>z"); // histogram x only if y>x
```

Als dritte Option können Darstellungsoptionen wie für die entsprechenden Histogramme angegeben werden:

```
nt.Draw("x", "1", "lp");
nt.Draw("x:y", "1", "colz");
```

Normalerweise ist das Binning der so erzeugten Histogramme fest vorgegeben, aber man kann auch zuerst ein Histogramm erzeugen und die Ausgabe dann in dieses Histogramm umlenken.

```
TH1F hist("hist", "histogram", 10, -2., 2.);
nt.Draw("x>>hist");
```

Das Histogramm kann man auch beim Füllen anlegen:

```
nt.Draw("x>>hist2(100,-1.,1.)");
```

11.1.3 Anzeige

Einzelne Einträge können mit der Methode `Scan` angesehen werden, wobei die ersten beiden Argumente analog zu `Draw` anzugeben sind:

```
nt.Scan("x:y:z", "x<10");
```

11.1.4 Speicherung in einer Datei

Ntupel und Trees werden wie andere Objekte auch in einer `.root`-Datei gespeichert:

```
TFile file("ntfile.root", "RECREATE");
TNtuple nt("nt","ntuple","t:x:y");

nt.Fill(...);

nt.Write();
```

Auch das Zurücklesen funktioniert wie mit jedem anderen Objekt:

```
TFile file("ntfile.root", "READ");
TNtuple *nt;
file.GetObject("nt", nt);
if (nt==NULL) {
  cerr << "unable to access ntuple" << endl;
  exit(-1);
}

nt->Draw(...);
```

11.2 Trees

Ntupel sind zwar sehr einfach zu verwenden, bieten aufgrund der Beschränkung auf `float`s weniger Fleximilität als Trees im Allgemeinen.

Im Gegensatz zu Ntupeln muss bei Trees die Verwaltung der einzelnen Elemente, genannt Blätter oder Leafs manuell erfolgen. D.h. für jedes zu speichernde Blatt muss ein Branch erzeugt werden. Dabei muss für jeden Branch ein Speicherplatz zur Verfügung stehen, über den die Ein- und Ausgabe abgewickelt wird.

Die einzelnen Branches werden mit der Methode `TTree::Branch(char* name, void* data, char* leaflist)` angelegt. Dabei steht:

- `name`: für den Namen des Branches

- `data`: zeigt auf einen Speicherbereich, der zum Lesen und Schreiben verwendet wird, und der groß genug für die Daten dieses Branches sein muss
- `leaflist`: ist eine Liste der elementaren Blätter dieses Branches und ihrer Datentypen. Jedes Feld ist mit Name, einem Schrägstrich und dem Datentyp (z.B. 'I'—32bit signed int 'F'—float) aufgeführt. Die Felder sind durch Doppelpunkte getrennt.

```
TFile f("tree.root","recreate");
TTree t1("t1","a simple Tree with simple variables");

Float_t px, py, pz;
Int_t ev;

t1.Branch("px",&px,"px/F");
t1.Branch("py",&py,"py/F");
t1.Branch("pz",&pz,"pz/F");
t1.Branch("ev",&ev,"ev/I");

for (...) {

  px = ...;
  py = ...;
  pz = ...;
  ev = ...;

  t1.Fill();
}
```

11.2.1 Branches mit mehreren Blättern

Alternativ kann man auch mehrer Elemente in einem Branch zusammenfassen. Dabei müssen die Elemente im Speicher an aufeinanderfolgenden Adressen stehen, z.B. indem man einen C-struct definiert:

```
struct {
  Float_t px;
  Float_t py;
  Float_t pz;
} p;
```

```
t1.Branch("p",&p,"px/F:py/F:pz/F");
```

11.2.2 Eintragsweise Analyse

Für komplexere Analysen kann es sinnvoll sein, die Blätter für einen Eintrag in den Speicher zu laden, um dann beliebige Analyseschritte durchzuführen.

Dies wird mit der Methode `GetEntry(int n)` erreicht, die den Eintrag in vorher festgelegte Speicherbereiche liest. Die Anzahl der Einträge kann mit `GetEntries` abgefragt werden:

```
tr->SetBranchAddress("px",&px);
tr->SetBranchAddress("py",&py);
tr->SetBranchAddress("pz",&pz);
tr->SetBranchAddress("ev",&ev);

for (int i=0; i<tr->GetEntries(); i++) {

  tr->GetEntry(i);

  cout << i << ": " << px << "/" << py << "/" << pz << endl;
}
```

11.2.3 Branches in separaten Dateien

Man kann einzelne Branches auch in einer anderen Datei speichern. Dies ist sinnvoll, wenn diese Branches nur selten gebraucht werden. Dann kann man die häufig benötigten Daten auf Festplatte vorhalten, während die unbenutzten Daten auf Bändern o.ä. liegen können.

Dazu muss den Branches vor dem Lesen und Schreiben eine andere Datei als Speicherort zugewiesen werden:

```
TFile *fbranch = ...
tree->GetBranch("branchname")->SetFile(fbranch);
```

11.2.4 Friends

Ein Nachteil bei Branches in verschiedenen Datein ist, dass nur in der Hauptdatei ein Verzeichniseintrag für den Tree ist. Die anderen Dateien scheinen auf den ersten Blick leer.

Ein Alternative ist das speichern eines vollständigen Trees in der zweiten Datei. Dieser zweite Tree kann beim Lesen wieder mit dem ersten Tree verbunden werden.

Beim Füllen werden 2 getrennte Trees erzeugt, wobei zu beachten ist, welche Daten in welche Datei geschrieben werden.

Zum Lesen wird der zweite Tree dann zum Freund des ersten Trees erklärt:

```
main_tree->AddFriend("friend_tree", "friend_file.root");
```

Dabei ist `friend_tree` der Name des Trees in der Datei `friend_file.root`.

11.2.5 Chains

Hat man in mehreren Dateien identische Trees angelegt, so können diese Trees für die Analyse der Daten zu einem Tree zusammengefügt werden. Dieser Mechanismus wird von der Klasse `TChain` implementiert.

```
TChain chain("tr");
chain.Add("split00.root");
chain.Add("split01.root");
```

Die Trees in den einzelnen Dateien wurden vorher als eigenständige Trees erzeugt. Die `TChain` wird nur beim Lesen benutzt.

11.2.6 Klassen in Trees

Auch das direkte Speichern von Klassen ist möglich. So kann eine objektorientierte Analyseumgebung sehr gut mit ROOT-Dateien kombiniert werden.

Soll ein Klasse in einer Datei gespeichert werden, so wird die Methode `Branch` mit einem Zeiger auf das zu speichernde Object aufgerufen.

```
person *p = new person;
tree.Branch("person", &p);
```

Wird der Tree wieder gelesen, so stehen für die Analyse die vollständigen Klassen mit allen Methoden wieder zur Verfügung.

```
person *p = new person;

TBranch *branch = tr->GetBranch("person");
branch->SetAddress(&p);
```

11.2.7 TClonesArray

Oft müssen mehrere Objekte unterhalb eines u"bergeordneten Objektes gespeichert werden, z.B. Tracks in einem Event. Die effizienteste Methode, dies zu tun sind `TClonesArrays`.
Ein TClonesArray

- enthält mehrere Objekte derselben Klasse
- die Elemente müssen von `TObject` erben
- der Speicherbereich muss nur einmal alloziert werden
- die Elemente können als einzelne Branches eines Trees abespeichert werden

Anhang A

Literatur

Homepage der Vorlesung
`http://qgp.uni-muenster.de/~diete_00/teaching/ss10/root/`

Die Programmiersprache C

C++ für C-Programmierer

Skripte des RRZN Hannover, erhältlich zum Selbstkostenpreis im ZIV, Einsteinstraße 60, Zimmer 104

Knappe Einführung in die Programmierung mit C/C++, in der alle wesentlichen Konzepte erkärt sind. Eignet sich auch gut als Nachschlagewerk, das man während des Programmierens immer wieder heranziehen kann.

B. Stroustrup:
Programming: Principles and Practice Using C++
Einführung in die Programmierung mit C++

Sehr neues Buch von Bjarne Stroustrup, dem Erfinder von C++. Das Buch wurde als Lehrbuch konzipiert und basiert auf Kursen an der Texas A&M University.

B. Stroustrup:
The C++ Programming Language
Die C++-Programmiersprache

Die C++ Bibel. Sehr gutes Nachschlagewerk, in dem C++ mit sehr vielen Details und Hintergrundinformationen auf relativ engem Raum erkärt ist. Als Einführung weniger geeignet, da die meisten Themen sehr knapp behandelt werden.

ROOT Homepage
`http://root.cern.ch/`

`http://root.cern.ch/drupal/content/users-guide`
Offiziele Seite von ROOT. Hier gibt es alles für ROOT: den User Guide, Tutorials, Beispielprogramme, Software-Download, Bug Reports.