EINFÜHRUNG IN C++

S. BARTELS, 20.11.2018

1.1. **Struktur.** Die Programmiersprache C++ ist eine Compiler-basierte Sprache, das heißt mit einem Texteditor wie *emacs* oder *kate* erstellte Programme werden mit Hilfe des Compilers in einen maschinenlesbaren Code übersetzt. Damit dies fehlerfrei funktioniert, müssen Programme nach einem vorgegebenen Rahmen geschrieben werden. Ein Programm der Sprache C++ beginnt mit der Einbindung benötigter vordefinierter Routinen, die in Bibliotheken und Klassen bereitgestellt werden, wie mathematischer Funktionen oder Ein- und Ausgabefunktionen. Es folgen optional selbstdefinierte Funktionen und am Ende steht das mit `main()` beginnende Hauptprogramm. Im Hauptprogram stehen Variablendefinitionen und Kommandos wie beispielsweise Wertzuweisungen und Funktionsaufrufe. Das linksseitig stehende Programm in Abbildung 1 zeigt ein einfaches Beispiel, in dem in einer Unterroutine das Quadrat einer Zahl berechnet wird. Diese wird vom Hauptprogramm aus mit einem Argument aufgerufen. Ist das Programm als Textdatei unter dem Namen `comp_square.cc` abgespeichert, so kann es im selben Verzeichnis mit dem Unix-Kommando

```
g++ comp_square.cc -o comp_square.out
```

kompiliert werden. Mit dem Unix-Kommando

```
./comp_square.out
```

wird das Programm gestartet.

```cpp
// comp_square.cc
#include <iostream>
#include <cmath>
double square(double x){
  return pow(x,2.0);
}
int main(){
  double x, y;
  x = 3.8;
  y = square(x);
  std::cout << "square is ";
  std::cout  << y << "\n";
}
```

```cpp
// simple_loop.cc
#include <iostream>
int main(){
  int i;
  for (i=0; i<5; i=i+1){
    std::cout << i << "\n";
  }
  std::cout << "\n";
  if (i==5){
    std::cout << "i is 5 \n";
  }
}
//
```

ABBILDUNG 1. Elementare Programme in C++.

1.2. **Klassen.** Die Klasse `iostream` stellt Routinen zur Ein- und Ausgabe zur Verfügung, während in der Klasse `cmath` Implementationen elementarer mathematischer Funktionen realisiert sind. Die artihmetischen Grundoperationen `+,-,*,/` können ohne die Einbindung von Bibliotheken verwendet werden. Zur Ausgabe von Text und Zahlen wird das Kommando `std::cout << "text \n"` beziehungsweise `std::cout << x` verwendet, wobei `\n` einen Zeilenumbruch bewirkt. Das Einlesen von Werten für eine Variable erfolgt mit `std::cin >> x;`. Durch die Anweisung

```
using namespace std;
```

können die Zusätze `std::` vermieden werden.

`cout`, `cin`	Aus- und Eingabe von Text und Variablen
`\n`, `endl`	Erzeugung eines Zeilenumbruchs
`/*...*/`, `//`	Mehr- und einzeilige Kommentare
`cos`, `sin`, `tan`	Trigonometrische Funktionen
`exp`, `log`, `log10`	Exponentialfunktion und Logarithmen
`pow`, `sqrt`	Potenz und Quadratwurzel
`floor`, `ceil`, `fabs`	Runden auf ganze Zahlen und Betragsfunktion

TABELLE 1. Ein- und Ausgabefunktionen, Kommentare sowie elementare mathematische Funktionen.

1.3. **Typen.** Jede Variable muss in C++ deklariert werden, das heißt vor ihrer Verwendung wird festgelegt, ob in ihr etwa eine ganze oder eine Gleitkommazahl das heißt Werte vom Typ *integer* oder *double* abgelegt werden sollen. Auch die Verwendung von Zahlen und arithmetischen Operationen ist mit Typen verbunden, beispielsweise wird der Ausdruck 2 als Variable vom Typ *integer* interpretiert, während 2. als Variable vom Typ *double* verwendet wird. Die Operation 2/3 wird in C++ als binäre Operation des höherwertigen Variablentyps ausgeführt, das heißt

$$2/3 = 0, \quad 2./3. \approx 0.\overline{6}, \quad 2/3. \approx 0.\overline{6}, \quad 2./3 \approx 0.\overline{6}.$$

Variablen können beispielsweise mit `x = (double)a` umgewandelt werden. Listen und Matrizen fester Größe werden als *Arrays* bezeichnet und können über `double x[n]` oder `double A[m][n]` initialisiert werden. Die Indizierung der Einträge von Arrays beginnt mit 0. Eine falsche Indizierung von Arrays führt im Allgemeinen nicht zu einer Fehlermeldung und muss vom Programmierer ausgeschlossen werden. Bei der Deklaration einer Variablen kann bereits ein Wert zugewiesen werden, sofern es sich nicht um ein Array handelt, dessen Größe durch eine Variable definiert ist.

1.4. **Kontrollanweisungen.** In C++ können Fallunterscheidungen, Wiederholungen und Aufzählungen in den in Abbildung 2 dargestellten Formaten realisiert werden. Dabei steht `cond` für eine logische Bedingung, die über eine logische Operation wie `a<b` definiert sein kann, während `statement` für

`int`	Ganzzahlige Maschinenzahlen
`float`, `double`	Gleitkommazahlen einfacher und doppelter Genauigkeit
`bool`	Boolesche Variablen mit Werten 0 und 1

TABELLE 2. Variablentypen in C++.

eine Liste von Kommandos steht, die von geschwungenen Klammern eingefasst sind. Die Ausdrücke `init` und `step` stehen für eine Initialisierung wie `i=0` und eine Anweisung der Art `i=i+1`, deren Ausführung so lange wiederholt wird, wie die Bedingung `cond`, beispielsweise `i<5`, als wahr ausgewertet wird. Dabei wird zunächst `init` ausgeführt, dann `cond` überprüft, anschließend der Kommandoblock `statement` abgearbeitet und schließlich `step` ausgeführt, bevor wiederum die Bedingung `cond` ausgewertet wird und dieser Vorgang wiederholt wird, bis `cond` falsch ist. Gelegentlich ist auch die Verwendung der `do while`-Schleife sinnvoll, bei der die Bedingung im Unterschied zur `while`-Schleife nach statt vor der Ausführung der Kommandos überprüft wird.

```
if (condA) statementA else if (condB) statementB else statementC
while (cond) statement
for (init; cond; step) statement
```

ABBILDUNG 2. Kontrollstrukturen in C++: Fallunterscheidung, Wiederholung und Aufzählung.

1.5. **Logische Ausdrücke und Inkremente.** Zur Formulierung logischer Bedingungen stehen boolesche Variablen, binäre Operationen zum Vergleich von Maschinenzahlen, die logischen Verknüpfungen *und*, *oder* sowie die Negation zur Verfügung. Vergleiche stehen dabei in Klammern also beispielsweise `(a<b)`. Gleitkommazahlen werden nur bis auf Maschinengenauigkeit verglichen und aufgrund möglicher Störungen ist ein Test auf exakte Gleichheit zweier Gleitkommazahlen wenig sinnvoll. Das Kommando `i=i+1` kann in C++ durch `i++` oder `++i` ersetzt und in arithmetischen oder logischen Ausdrücken verwendet werden. Im Fall `i++` wird die Variable zunächst erhöht und anschließend der Ausdruck ausgewertet, während bei Verwendung von `++i` zunächst die Variable erhöht wird. Die logischen Ausdrücke `(i++==1)` or `(++i==1)` führen also zu unterschiedlichen Wahrheitswerten.

1.6. **Funktionen.** Funktionen können entweder eine oder keine Variable zurückgeben. Wird ein Wert zurückgegeben, so steht der Typ des Funktionswerts vor dem Funktionsnamen, andernfalls wird `void` verwendet. Auf den Funktionsnamen folgt in Klammern eine Liste von Argumenten. Argumente einfachen Typs wie `double` und `int` werden mittels *call by value*

`==`, `!=`, `>`, `>=`, `<`, `<=`	Logischer Vergleich von Maschinenzahlen
`&&`, `||`, `!`	Logische Verknüpfungen *und*, *oder* sowie Negation
`i++`, `++i`, `i--`, `--i`	Prä- und Postinkrement sowie -dekrement
`b+=x`	Kurzform für `b=b+x`

TABELLE 3. Logische Operationen sowie Inkrement- und Dekrementfunktionen.

behandelt, das heißt sie werden in eine lokale Variable kopiert. Die entsprechende aufrufende Variable im Hauptprogramm bleibt dabei unverändert. Arrays sind nicht als Rückgabewert einer Funktion zugelassen. Sie werden an Funktionen daher als Argumente mittels *call by reference* übergeben und dabei als globale Variablen von dem Unterprogramm verändert. In dem in Abbildung 3 links gezeigten Programm wird das Array `x` an die Funktion `mod_vector` übergeben und dort unter dem Namen `vec` verwendet. Nach dem Durchlauf der Funktion sind die Werte des Arrays `x` verändert. Wichtig ist hierbei die Verwendung des Sterns bei der Deklarierung des Arguments der Funktion.

```cpp
// static_array.cc
#include <iostream>
using namespace std;
void mod_vector(double* vec){
  vec[0] = 2.0;
  vec[1] = 1.0;
}
int main(){
  const int n = 2;
  double x[n];
  x[0] = 1.0;
  x[1] = 2.0;
  mod_vector(x);
  cout << "x[0] = " << x[0];
  cout << "\n";
  cout << "x[1] = " << x[1];
  cout << "\n";
}
```

```cpp
// functions.cc
#include <iostream>
using namespace std;
void fun_1(double z){
  z = z+1.0;
}
void fun_2(double* z){
  *z = *z+1.0;
}
int main(){
  double x = 1.0;
  fun_1(x);
  cout << "x = " << x << "\n";
  fun_2(&x);
  cout << "x = " << x << "\n";
}
//
//
```

ABBILDUNG 3. Übergabe von Arrays sowie einfachen Variablen und Pointern an Funktionen.

1.7. **Pointer.** Ein Pointer oder Zeiger ist eine Variable, die die Adresse eines Abschnitts im Speicher enthält. Der Pointer erlaubt es, den Inhalt des entsprechenden Speicherabschnitts auszulesen oder zu verändern. Die Größe des Speicherabschnitts hängt davon ab, ob dort eine Gleitkommazahl oder

ganzzahlige Maschinenzahl abgelegt werden soll. Ist `ptr` ein Pointer, so ist der Wert der Variable, die unter der in `ptr` enthaltenen Adresse zu finden ist, gegeben durch die gewöhnliche Variable `*ptr`. Umgekehrt wird für eine gewöhnliche Variable `var` durch `&var` ein Pointer definiert, der die Adresse des entsprechenden Speicherplatzes enthält. Deklariert wird ein Pointer beispielsweise mittels `double* ptr`. Übergibt man die Adresse einer Variable, das heißt den entsprechenden Pointer, an eine Funktion, dann wird diese Variable mittels *call by reference* behandelt, so dass der Inhalt der Variablen mit globalen Auswirkungen manipuliert wird. Die oben beschriebene Übergabe von Arrays an Funktionen folgt gerade diesem Prinzip. In dem in Abbildung 3 rechts gezeigten Programm verändert die Funktion `fun_1` den Wert der Variablen `x` des Hauptprogramms nicht, während die Funktion `fun_2` deren Wert erhöht. Ein in einer Funktion definierter Pointer kann als Rückgabewert der Funktion verwendet werden. In diesem Fall muss die Funktion beziehungsweise ihr Wert mittels `double*` deklariert werden. Die Übergabe von Variablen an Funktionen mittels Pointern vermeidet beispielsweise das Umkopieren großer Daten.

1.8. **Dynamische Arrays.** Die oben beschriebene Verwendung von Arrays stößt an Grenzen, wenn die Dimension der Arrays erst im Laufe des Programmdurchlaufs bestimmt wird. Die Klasse `vector` stellt Hilfsmittel zur Deklaration von Vektoren mit Einträgen bestimmter Typen und deren Manipulation zur Verfügung. Die Länge so definierter Vektoren kann dann beliebig verändert werden. Um Vektoren beispielsweise mit Gleitkommazahl-Einträgen übersichtlich in Programmen verwenden zu können, sollte ein neuer Variablentyp definiert werden, wie beispielsweise der Typ `doubleVec`:

```
typedef typename std::vector<double> doubleVec;
```

In einer entsprechenden Variablendeklaration können dann optional die Länge des Vektors und Einträge angegeben werden. Durch

```
doubleVec x(5,1.);
```

wird ein Vektor `x` der Länge 5 mit Gleitkommazahleinträgen, die anfangs auf Eins gesetzt sind, definiert. Die Länge und die Einträge können mit den in Tabelle 4 gezeigten Methoden verändert werden.

`x.size()`	Gibt die Länge des Vektors `x` an
`x.resize(n)`	Verändert die Länge eines Vektors
`x.push_back(val)`	Hängt den Eintrag `val` an einen Vektor an
`x.pop_back()`	Löscht das letzte Element eines Vektors

TABELLE 4. Methoden zur Manipulation einer Variable eines Vektortyps.

```
// dynamic_vectors.cc
#include <iostream>
#include <vector>
typedef typename std::vector<double> doubleVec;
doubleVec scan_vector(doubleVec x){
  for (int i=0; i<x.size(); ++i){
    std::cout << "x[" << i << "] = ";
    std::cin >> x[i];
  }
  std::cout << "\n";
  return(x);
}
void print_vector(doubleVec x){
  for (int i=0; i<x.size(); ++i){
    std::cout << "x[" << i << "] = " << x[i] << "\n";
  }
  std::cout << "\n";
}
int main(){
  int dim = 5;
  doubleVec y(dim,1.);
  print_vector(y);
  std::cout << "dim = \n";
  std::cin >> dim;
  y.resize(dim);
  y = scan_vector(y);
  print_vector(y);
}
```

ABBILDUNG 4. Ein- und Ausgabe von Vektoren beliebiger Länge.

1.9. **Arbeiten mit Matrizen.** Eine Matrix $A \in \mathbb{R}^{m\times n}$ kann mit einem Vektor $\widehat{A} \in \mathbb{R}^{mn}$ identifiziert werden, indem man die Spalten von A untereinander in einen Vektor schreibt. Es gilt, bei Numerierung der Einträge mit den Indizes $i = 0, 1, \dots, m-1$ und $j = 0, 1, \dots, n-1$, dass

$$A_{ij} = \widehat{A}_{i+jm}.$$

Mit dieser Identifikation lassen sich Matrizen in C++ wie Vektoren behandeln. Gelegentlich ist es übersichtlicher, Matrizen als zweidimensionale Arrays zu behandeln. Ist die Größe von vornherein bekannnt, so kann man sie wie einfache Variablen verwenden. Die Übergabe an Funktionen erfolgt dann jedoch mittels *call-by-reference*, wie in dem in Abbildung 5 gezeigten Programm.

1.10. **Zeitmessung, Speichern und Pakete.** Die Bibliothek `time.h` stellt den Variablentyp `clock_t`, das Kommando `clock()`, und die Konstante `CLOCKS_PER_SEC` zur Verfügung, mit denen Laufzeitmessungen durchgeführt werden können. Deren Verwendung ist in Abbildung 6 illustriert.

```
// matrix.cc
#include <iostream>
const int m = 3;
const int n = 2;
void mod_matrix(double mat[m][n]){
  mat[0][0] = 7.0;
  mat[2][1] = 8.0;
}
int main(){
  double A[m][n] = {{1.,2.},{3.,4.},{5.,6.}};
  mod_matrix(A);
  std::cout << "A[0][0] = " << A[0][0] << "\n";
  std::cout << "A[2][1] = " << A[2][1] << "\n";
}
```

ABBILDUNG 5. Verwendung zweidimensionaler Arrays.

Zum Speichern von Daten können Methoden aus der Klasse `fstream` verwendet werden. Mit dem Variablentyp `fstream` und der Methode `open` kann ein Pointer auf eine Datei definiert werden, in die dann mit dem Umleitungsoperator `<<` geschrieben wird. Ist das Schreiben beendet, so muss die Datei mit `close` geschlossen werden. Das in Abbildung 6 rechts gezeigte Beispielprogramm speichert einen Vektor in der Datei `var.dat` ab. Diese kann von MATLAB mit dem Kommando `load var.dat` ausgelesen werden und weist die Werte des Vektors der Variablen `var` zu.
Die Pakete `BLAS` und `LAPACK` stellen Implementationen numerischer Verfahren beispielsweise zur Lösung linearer Gleichungssysteme und Eigenwertaufgaben bereit.

```cpp
// runtime.cc
#include <iostream>
#include <time.h>
int main(){
  double diff, dt, x = 0.33;
  clock_t t1, t2;
  t1 = clock();
  for (int i=0; i<100000; i++){
    x*x;
  }
  t2 = clock();
  diff = double(t2-t1);
  dt = diff/CLOCKS_PER_SEC;
  std::cout << "runtime = ";
  std::cout << dt << "\n";
}
```

```cpp
// save_data.cc
#include <fstream>
int main(){
  std::fstream file;
  double x[3] = {0.,1.,3.};
  file.open("var.dat",
     std::ios::out);
  if (file.good()){
    for (int i=0; i<3; i++){
      file << x[i] << "\n";
    }
  }
  file.close();
}
//
//
```

ABBILDUNG 6. Laufzeitmessung und Speichern von Daten.

```
// lu_solution.cc
#include <iostream>
const int n = 3;
void lu_crout(double A[n][n]){
  double sum;
  for (int i=0; i<n; i++){
    for (int k=i; k<n; k++){
      sum = 0.0;
      for (int j=0; j<=i-1; j++){sum = sum+A[i][j]*A[j][k];}
      A[i][k] = A[i][k]-sum;
    }
    for (int k=i+1; k<n; k++){
      sum = 0.0;
      for (int j=0;j<=i-1;j++){sum = sum+A[k][j]*A[j][i];}
      A[k][i] = (A[k][i]-sum)/A[i][i];
    }
  }
}
void solve_lower_normalized(double L[n][n], double b[n],
                double y[n]){
  double sum;
  for (int j=0; j<n; j++){
    sum = 0.0;
    for (int k=0; k<=j-1; k++){sum = sum+L[j][k]*y[k];}
    y[j] = b[j]-sum;
  }
}
void solve_upper(double U[n][n], double y[n], double x[n]){
  double sum;
  for (int j=n-1; j>=0; j--){
    sum = 0.0;
    for (int k=j+1; k<n; k++){sum = sum+U[j][k]*x[k];}
    x[j] = (y[j]-sum)/U[j][j];
  }
}
int main(){
  double x[n], y[n];
  double A[n][n] = {{2.,-1.,0.},{-1.,2.,-1.},{0.,-1.,2.}};
  double b[n] = {1.,1.,1.};
  lu_crout(A);
  solve_lower_normalized(A,b,y);
  solve_upper(A,y,x);
  for (int i=0; i<n; i++){
    std::cout << "x[" << i << "] = " << x[i] << "\n";
  }
}
```

ABBILDUNG 7. LU-Verfahren in C++.

```
// neville_scheme.cc
#include <iostream>
#include <vector>
typedef typename std::vector<double> doubleVec;
const int n = 3;
double neville_recursive(double z, doubleVec x, doubleVec y,
             int i, int j){
  if (j==0){
    return y[i];
  }
  else{
    return ((z-x[i])*neville_recursive(z,x,y,i+1,j-1)
     -(z-x[i+j])*neville_recursive(z,x,y,i,j-1))/(x[i+j]-x[i]);
  }
}
double neville_forward(double z, doubleVec x, doubleVec y){
  double P[n+1][n+1];
  for (int i=0; i<=n; i++){
    P[i][0] = y[i];
  }
  for (int j=1; j<=n; j++){
    for (int i=0; i<=n-j; i++){
      P[i][j] = ((z-x[i])*P[i+1][j-1]-(z-x[i+j])*P[i][j-1])/
    (x[i+j]-x[i]);
    }
  }
  return P[0][n];
}
int main(){
  int N = 20;
  doubleVec x(n+1), y(n+1);
  doubleVec z(N+1), w_rec(N+1), w_for(N+1);
  x[0] =-1.0; x[1] =-1.0/3; x[2] = 1.0/3; x[3] = 1.0;
  y[0] =-1.0; y[1] = 1.0;   y[2] =-1.0;   y[3] = 1.0;
  for (int k=0; k<=N; k++){
    z[k] = -1.0+2.0*(double)k/N;
    w_rec[k] = neville_recursive(z[k],x,y,0,n);
    w_for[k] = neville_forward(z[k],x,y);
    std::cout << "w_rec = " << w_rec[k] << ", ";
    std::cout << "w_for = " << w_for[k] << "\n";
  }
}
```

ABBILDUNG 8. Neville-Verfahren in C++.

```
// implicit_euler.cc
#include <fstream>
#include <cmath>
#include <vector>
typedef typename std::vector<double> doubleVec;
double f(double t, double y){
  return cos(2.0*t)*pow(y,2.0);
}
double Phi(double t, double y_old, double y_new, double tau){
  return f(t+tau,y_new);
}
void save_solution(doubleVec y, int K){
  std::fstream file;
  file.open("sol.dat",std::ios::out);
  if (file.good()){
    for (int k=0; k<=K; k++){
      file << y[k] << "\n";
    }
  }
  file.close();
}
int main(){
  double y_0 = 1.0, T = 10.0, tau = 1.0/100.0, t_k;
  double z, z_new, diff, eps_stop = tau/10;
  int k, K = floor(T/tau);
  doubleVec y(K+1);
  y[0] = y_0;
  for (k=0; k<K; k++){
    t_k = k*tau;
    z = y[k];
    diff = 1.0;
    while (diff>eps_stop){
      z_new = y[k]+tau*Phi(t_k,y[k],z,tau);
      diff = fabs(z_new-z);
      z = z_new;
    }
    y[k+1] = z;
  }
  save_solution(y,K);
}
```

ABBILDUNG 9. Implizites Euler-Verfahren in C++.