Vorkurs Informatik: Erste Schritte der Programmierung mit C++

Arne Nägel, Andreas Vogel, Gabriel Wittum
Lehrstuhl Modellierung und Simulation
Goethe-Center for Scientific Computing
Goethe Universität Frankfurt a.M.

6. Oktober 2014

STARKER START ins Studium

Agenda

Literaturhinweise (Klassiker)

Kernighan, Brian W. ; Ritchie, Dennis M.: The C programming language. Prentice Hall, 1978

Stroustrup, Bjarne: The C++ programming language. Addison-Wesley, 1985 (online - neuere Auflage)

Literaturhinweise (Einsteiger und Fortgeschrittene)

- Stroustrup, Bjarne: Einführung in die Programmierung mit C++. Pearson Studium, 2010
- Willms, André: C++-Programmierung lernen : Anfangen, Anwenden, Verstehen. Addison-Wesley, 2008 . (online)
- Breymann, Ulrich: C++ : Einführung und professionelle Programmierung (9. Aufl). Hanser, 2007
  Breymann, Ulrich: Der C++-Programmierer : C++ lernen - professionell anwenden - Lösungen nutzen. Hanser, 2009
- Dieterich, Ernst-Wolfgang: C++. Oldenbourg, 2000 (online)

rundlegende Sprachelemen

Grundlegende Sprachelemente von C++ (Fortsetzung)

Referenzen

Eine **Referenz** ist ein Alias (Pseudoname) für eine Variable. Sie ist ein Verweis auf die Variable und kann genauso wie diese benutzt werden. Ein Variable hat dann mehrere Namen.

In C++ werden Referenzen über ein angehängtes &-Zeichen an den Datentyp realisiert.

```
...
int a = 1;
int& b = a; // Referenz von a
b = 3; // a ist nun 3
std::cout << "Wert von a: "<< a << std::endl;
...
```

const-Referenz

Referenzen können auch als konstant deklariert werden. Dann kann man den Wert über die Referenz nicht ändern.

```
...
int a = 1;
const int& b = a; // const-Referenz von a
a = 2; // Ok: a ist nicht const
b = 3; // Fehler: die Referenz ist konstant
...
```

Feld

Ein **Feld** fasst Daten desselben Typs zusammen. Es lässt sich als ein Vektor betrachten.

```
<Typ> <Bezeichner> [<Größe>];
```

```
int main(){
  int vFeld[5]; // Feld der Groesse 5
  vFeld[0] = 2; // Zugriff erstes Element
  vFeld[4] = 3; // Zugriff letztes Element
}
```

Ein Feld der Größe N wird von $0, \ldots, N-1$ gezählt.
Der Zugriff eines Felds ist über den Operator [ ].

Mehrdimensionale Felder

Es lassen sich auch mehrdimensionale Felder erzeugen.

```
{
  const int NumRows = 2;
  const int NumCols = 3;
  int a[NumRows][NumCols] = {{1,2,3},{4,5,6}};

  for(int i = 0; i < NumRows; ++i){
    for(int j = 0; j < NumCols; ++j){
      std::cout << a[i][j] << ' ';
    }
    std::cout << std::endl;
  }
}
```

Funktionen

Funktionen

Eine Funktion ...

- erledigt eine abgeschlossene Teilaufgabe
- macht das Programm übersichtlicher
- macht Codeabschnitt wiederverwendbar
- macht Funktionalität (z.B. als Bibliothek) auslagerbar

Definition:

```
<ReturnTyp> <FunktionsName> ( <Parameterliste> ) {
   <Anweisungen>
}
```

Deklaration/Definition einer Funktion

Die Kombination aus Funktionsname und Parameterliste wird **Signatur** genannt.

Eine Funktion muss genau einmal **definiert** werden. Die Definition einer Funktion besteht aus Signatur und Funktionsrumpf. Eine

Funktion kann beliebig oft **deklariert** werden. Die Deklaration einer Funktion besteht nur aus der Signatur.

Funktionsdefinition (Beispiel)

```
// Deklaration
int Add(int x, int y);

int main(){
  Add(3,4); // nutze die Funktion
}

// Definition
int Add(int x, int y){
 return x+y;
}
```

Merke: Damit eine Funktion verwendet werden kann, muss nur die Deklaration geben sein. Die Definition kann später (z.B. auch in einer anderen Datei) erfolgen.

Call by Value

Sind die Typen in der Parameterliste kein Zeiger oder Referenzen, so wird beim Aufruf die Variable **kopiert** (Call by Value).

```
void AddOne(int x){
  x += 1;
  std::cout << "x: " << x; // Ausgabe: 6
}

int main(){
  int a = 5;
  AddOne(a);
  std::cout << "a: " << a; // Ausgabe: 5
}
```

Das übergebene Objekt wird **nicht verändert**.

Call by Reference

Sind die Typen in der Parameterliste Referenzen, so wird beim Aufruf die Variable per **Referenz** übergeben (Call by Reference).

```
void AddOne(int& x){
  x += 1;
  std::cout << "x: " << x; // Ausgabe: 6
}

int main(){
  int a = 5;
  AddOne(a);
  std::cout << "a: " << a; // Ausgabe: 6 (!!)
}
```

Das übergebene Objekt wird **verändert**. Es hat in der Funktion nur einen anderen Namen.

Vorsicht bei Call by Reference

Werden Parameter per Reference (oder als Zeiger) übergeben, muss man sich bewusst sein, dass sie verändert werden können.

Möchte man keine Kopie machen (z.B. bei großen Objekten), diese jedoch nicht verändern lassen, so kann man sie als **`const` Referenz** übergeben. Der Compiler überprüft dann, ob dies tatsächlich der Fall ist. Der Aufrufer der Funktion kann sich also sicher sein, dass sein Objekt nicht verändert wird.

```
void AddOne(const int& x){
  x[0] += 1; // Fehler: Darf nicht veraendert werden
}

int main(){
  int a = 5
  AddOne(a);
}
```

Rückgabewerte

- Die `return`-Anweisung beendet die Funktion und gibt den Wert zurück, der hinter `return` angegeben wurde.
- Auch eine Funktion vom Typ `void` kann beendet werden. Verwenden Sie dazu `return;` (ohne weiteres Argument)
- Stellen Sie immer sicher, dass Ihre Funktionen sinnvolle Rückgabewerte liefern!

Rückgabewerte

Der Rückgabe einer Funktion kann auch by Value oder by Reference geschehen. Bei Rückgabe per Referenz muss man darauf achten, dass die Variable auch nach der Funktion noch existiert. Daher niemals lokale Variablen per Referenz zurückgeben!

```
int& Increment_Good(int& x){
  return ++x;
}

int& Increment_Bad(int x){
  return ++x;
}

int main(){
  int a = 1;
  std::cout << Increment_Good(a); // ok
  std::cout << Increment_Bad(a);  // Fehler: RÃ¼ckgabevariable ("x")
     existiert nicht mehr
}
```

Variablen in Funktionen

Für die Gültigkeits- und Sichtbarkeitsregeln für Variablen gelten diesselben wie üblich - eine Funktion ist ein Block.

Die Variablen der Parameterlist sind lokale Variablen (d.h. nur innerhalb des Blocks sichtbar).

Ausnahme: **Statische** Variablen. Ist eine Variable als `static` deklariert, so wird sie beim ersten Aufruf initialisiert und der Wert bleibt auch bei weiteren Aufrufen erhalten.

```
void print(){
  static int val = 1;
  std::cout << "val: " << val++; // Aufrufe: 1, 2, 3, ...
}
```

Rekursive Funktionen

- Indem eine Funktion sich selber aufruft, erzeugt man eine **Rekursion**
- Rekursive Funktionen können sehr nützlich sein (z.B. Türme von Hanoi)
- Ein Funktionsaufruf benötigt jedoch Zeit und Speicher!

Überladene Funktionen

Es kann mehrere Funktionen mit demselben Funktionsnamen geben. Dann muss sich jedoch die Parameterliste unterscheiden (Returntyp reicht nicht). Die Funktion wird **überladen** genannt.

```
int Add(int x, int y);

double Add(double x, double y);  // ok, anderer Typ
int Add(int x, int y, int z);    // ok, verschieden Zahl Parameter

double Add(int x, int y);        // Fehler: nur Returntyp anders
```

Default Parameter

Parameter einer Funktion können mit Defaultwerten deklariert werden. Diese werden verwendet, wenn die Parameter nicht angegeben werden.

```
bool IsSmall(double x, double tol = 1e-8){
  if(x < tol) return true; else return false;
}

int main(){
  std::cout << IsSmall(1e-15); // true
  std::cout << IsSmall(1e-15, 1e-18); // false
}
```

inline-Funktionen

Ein Funktionsaufruf kostet Zeit: Parameter müssen kopiert werden, das Programm springt an eine andere Stelle und nach der Funktion wieder zurück und der Stack muss angepasst werden. Diesen Aufwand möchte man gerne bei sehr kleinen Funktionen vermeiden.

Deshalb kann man eine Funktion als `inline` deklarieren. Dies schlägt dem Compiler vor, dass er diese Funktion direkt an der Stelle einsetzen soll und intern somit keinen Funktionsaufruf durchführen soll. Dies ist nur eine Empfehlung für den Compiler, wird jedoch oftmals vorgenommen.

```
inline int Add(int x, int y) { return x+y;}

int main(){
  std::cout << Add(3,4);  // inline Funktionsaufruf
}
```

Modulares Programmdesign

Modulares Programmdesign

Bei größeren Programmen stellt sich die Frage, wie das Programm sinnvoll in kleinere Teile aufgeteilt werden kann.

Grundsätzlich sollten verschiedene, nicht voneinander abhängige Funktionalitäten in **verschiedene Dateien** aufgeteilt werden. Dabei kann man durchaus ähnliche Funktionalität in einer Datei gruppieren.

Deklarationen sollten von der Implementierung abgespalten werden. Die Implementierung wird in einer *.cpp Datei umgesetzt, die Deklaration sollte in einer *.h Datei vorgenommen werden.

Andere Programmteile brauchen dann nur den sogenannten Header (*.h) einbinden, d.h. nur die Deklaration.

Beispiel

```
int Add(int x, int y);
```

add.h

```
#include "add.h"
int Add(int x, int y){
return x+y;
}
```

add.cpp

```
#include "add.h"
int main(){
std::cout << Add(5,6);
}
```

myProc.cpp

Vorteil:

- Programm ist modular aufgebaut
- Programmteile können einzeln übersetzt werden
- Implementierungen können ausgetauscht werden
- ilierzeiten werden reduziert

Include-Guards

Oftmals inkludieren Dateien viele andere *.h Dateien und auch Header inkludieren weitere Header. Damit am Ende nicht die Deklarationen oft kompiliert werden, hat es sich eingebürgert sogenannte **Include-Guards** zu verwenden. Diese sorgen dafür, dass jeder Header nur einmal inkludiert wird.

```
#ifndef __H__SOME_HEADER__
#define __H__SOME_HEADER__

// .. Deklarationen stehen hier

#endif // end __H__SOME_HEADER__
```

someHeader.h

Die Wahl der Include-Guards muss eindeutig sein.

Zeiger und dynamische Speicherverwaltung

Zeiger

Auf Variablen kann man nicht nur direkt, sondern auch über **Zeiger** (Pointer) zugreifen. Ein Zeiger zeigt auf die Adresse (Speicherstelle), an der die Variable gespeichert ist. Die Syntax ist:

```
<Typ>* <Bezeichner>;
```

Oder auch: `<Typ> *<Bezeichner>;`

Die Adresse einer Variablen bestimmt der **Adressoperator** "&":

```
&<Bezeichner>
```

Die Variable, auf die ein Zeiger zeigt, erhält man durch den **Dereferenzoperator** (Zugriffsoperator) "*":

```
*<Zeiger>
```

Zeiger (Beispiel)

```
{
  int i = 5, j = 10; // zwei Variablen vom Typ int
  int* p;  // ein Zeiger auf ein Variable vom Typ int (
    uninitialisiert)
  p = &i; // p zeigt auf i
  std::cout << p; // ... gibt die Speicheradresse aus, auf die p
    zeigt

  std::cout << *p; // ... gibt den Wert der Variablen i aus

  p = &j; // p zeigt nun auf j
  std::cout << *p; // ... gibt nun den Wert der Variablen j aus
}
```

NULL - Zeiger

Es gibt eine spezielle Zeigerkonstante 0 (NULL in C), welche auf die (hexadezimale) Speicheradresse 0x0 (= nil) verweist und bzgl. welcher eine Zeigervariable getestet werden kann. Diese dient dazu ungültigen Speicher darzustellen.

```
{
  double d = 1.345;
  double* p1 = 0, p2 = &d;

  if (p1 != 0)
    std::cout << *p1; // p1 ist kein gueltiger Zeiger, nicht (!)
    verwenden

  if (p2 != 0)
    std::cout << *p2; // p2 ist ein gueltiger Zeiger, kann verwendet
     werden
}
```

Zeigerarithmetik

Mit Zeigern lassen sich arithmetische Operationen durchführen:

- `==, !=, <, >, <=, >=`: Vergleiche bzgl. der Speicherstelle
- `+ / -`: Addition, Subtraktion (z.B. Speicherabstand)
- `p + N` := `p + N*sizeof(<typ>)`
- `++, --`: Inkrement, Dekrement

Zeiger und Felder

Ein Feld nutzen linearen Speicher, d.h. ein im Index nachfolgendes Element ist physisch im nachfolgenden Speicherbereich abgelegt.

Zeigervariablen lassen sich als Feldbezeichner nutzen und umgekehrt.

```
{
  int a[3] = {4,5,6};
  int* p;

  p = &a[0];          // p zeigt nun auf den Anfang des Felds
  std::cout << p[1]; // gibt den Wert an der 2ten Position des Felds

  std::cout << a[1] << p[1] << *(a+1) << *(p+1); // immer dasselbe
}
```

Call by Pointer

Sind die Typen in der Parameterliste Zeiger, so wird beim Aufruf die Variable die **Speicherstelle** übergeben.

```
void AddOne(int* x){
  *x += 1;
  std::cout << "x: " << *x; // Ausgabe: 6
}

int main(){
  int a = 5;
  AddOne(&a);
  std::cout << "a: " << a; // Ausgabe: 6 (!!)
}
```

Das übergebene Objekt wird **verändert**. Es hat in der Funktion nur einen anderen Namen (analog zu Call by Reference).

Übergabe von Feldern

Auch Felder können übergeben werden.

```
void AddOne(int[] x){
  x[0] += 1;
  std::cout << "x: " << x[0]; // Ausgabe: 6
}

int main(){
  int a[20]; a[0] = 5
  AddOne(a);
  std::cout << "a: " << a[0]; // Ausgabe: 6 (!!)
}
```

Exkurs: Zeiger auf Funktionen

Funktionen sind ebenfalls physikalisch im Speicher abgelegt. Man kann sich folglich einen Zeiger auf eine Funktion halten.

```
int max(int a, int b) {if(a>b) return a; else return b;}
int min(int a, int b) {if(a<b) return a; else return b;}

{
  int (*pF)(int,int);  // pF ist ein Zeiger auf eine Funktion

  pF = &max;
  std::cout << (*pF)(3,4); // gibt Maximum

  pF = &min;
  std::cout << (*pF)(3,4); // gibt nun Minimum
}
```

Exkurs: Zeiger auf Konstanten und konstante Zeiger

Auch Zeiger können `const` sein. Dies bedeutet, dass der Zeiger (d.h. die Speicherstelle, auf die er zeigt) nicht verändert werden kann. Dies muss man unterscheiden von Zeigern auf konstante Variablen. Bei diesen ist der Zeiger veränderbar, aber der hinterliegende Wert ist konstant.

```
{
  const int a = 1;
  int b = 2;
  const int* p1 = &a; // ok, variabler Zeiger, Wert nicht
    veraenderbar
  const int* p2 = &b; // ok, variabler Zeiger, Wert nicht
    veraenderbar
  int* p3 = &a;       // Fehler: int* auch const int

  int* const p4 = &b;    // ok, Konstanter Zeiger, Wert veraenderbar
  *p4 = 5;               // ok, Zeiger unveraendert, Wert geaendert
  p4 = &a;               // Fehler: Zeiger (Speicherstelle) nicht
    veraenderbar
```

Speicherverwaltung

Speicher selbst anlegen

Bisher: Speicher für Variablen wird innerhalb des Blocks angelegt und am Ende wieder freigegeben.

Man kann Speicher auch selber verwalten. Der Speicher wird dann dynamisch allokiert und **muss** vom Benutzer selbst wieder freigegeben werden. (keine Garbagecollection)

Allokieren: `<Typ>* p = new <Typ>;`
Freigeben: `delete p;`

Allokieren: `<Typ>* p = new <Typ>[<Size>];`
Freigeben: `delete[] p;`

Allokation (Beispiel)

```
{
 int* a;
 {
   a = new int[10]; // Speicher wird angelegt
 }
 a[2] = 5; // Speicher immer noch gÃ¼ltig

 delete[] a; // Speicher wird freigegeben
}
```

Allokation (Achtung)

```
{
 {
   int* a = new int[10]; // Speicher wird angelegt
 }

 // Zeiger a nicht mehr vorhanden, aber Speicher weiter belegt
}
```

Der Speicher **muss von Benutzer** wieder freigegeben werden!

Exkurs: Iterative und rekursive Prozesse