Informatik II

Oliver Jack

Fachhochschule Jena
Fachbereich Elektrotechnik und Informationstechnik

Sommersemester 2010

Inhalt

Lernziele dieser Vorlesung

Zeichenketten

Vektoren

Funktionen
- Definition und Deklaration
- Polymorphie und Gültigkeitsbereich
- Argumentübergabe
- Operatoren

Zusammenfassung

Vorlesung 4. Datentypen, Funktionen und Operatoren

Lernziele dieser Vorlesung

Zeichenketten

Vektoren

Funktionen
- Definition und Deklaration
- Polymorphie und Gültigkeitsbereich
- Argumentübergabe
- Operatoren

Zusammenfassung

4. Datentypen, Funktionen und Operatoren

Vorige Vorlesung

- Übergabe von Parametern an Programme
- Standardschema zur Bearbeitung von Programmparametern
- Ein- Und Ausgaben
- Lesen und Schreiben von Dateien

Heutige Vorlesung

- Spezielle Datenstruktur Zeichenkette (string)
- Vektoren und die Standard Template Library (STL)
- Funktionen

Lernziele

- Kenntnis der Datenstruktur string
- Verständnis von Vektoren
- Kenntnis des Funktionsmechanismus in C++
- Verständnis verschiedener Argumentübergabemechanismen
- Elementares Verständnis von Rekursion
- Kenntnis von Funktionsoverloading und Operatoren in C++

Zeichenketten (String): Wiederholung C-Strings

- Es gibt in C++ keinen Datentyp für Zeichenketten
- Zeichenketten sind Felder vom Typ `char`
- Vereinfachte Syntax: Zeichenkette wird in Anführungszeichen (") eingeschlossen
- Besonderheit: letzte Zeichen ist das Null-Zeichen (\0), wird automatisch angehängt
- Für eine Zeichenkette `string` der Länge n muss ein Feld der Länge $n+1$ deklariert werden (`char string[`$n+1$`]`)

"Hallo!"	entspricht	H	a	l	l	o	!	\0
"%$#^*"	entspricht	%	$	#	^	*	\0	
""	entspricht	\0						

String-Funktionen: Wiederholung C-Strings

`#include <string>`	
Funktion	**Beschreibung**
`strlen(`*string*`)`	gibt die Länge von *string* zurück, dabei wird das abschließende Null-Zeichen \0 nicht mitgezählt.
`strcmp(`*string1*`, `*string2*`)`	vergleicht *string1* und *string2*; der Rückgabewert ist 0, falls sie gleich sind. Ist *string1* lexikographisch größer als *string2*, wird eine positive Zahl zurückgegeben, ist *string1* lexikographisch kleiner als *string2*, eine negative Zahl.
`strcat(`*string1*`, `*string2*`)`	hängt *string2* an das Ende von *string1* an, *string2* bleibt unverändert.
`strcpy(`*string1*`, `*string2*`)`	kopiert den Inhalt von *string2* in *string1*.

Strings: C++...

- Standardbibliothek stellt einen `string`-Typ zur Verfügung.
- Operationen, exemplarisch

```
#include <string>

string s1 = "Hallo"; // Initialisierung
string s2 = "Welt";  // dito
string s3 = s1 + s2; // Konkatenation
s3 += '!'; // Ausrufzeichen anhaengen
string name = "Hans␣Maier";
string nachname = name.substr(5,5); // "Maier"
name.replace(0,3,"Herbert"); // "Herbert Maier"
```

- Vorteil gegenüber `char[n]`: flexibler, in Größe änderbar, z. B. durch Operator +

. . .Strings: C++. . .

- Weitere Operationen
- Finden, Länge

```
#include <string>

string s1, s2, s3;

cout << "Bitte␣zwei␣Worte␣eingeben:␣";
cin >> s1 >> s2;
cout << "s1:␣" << s1 << endl;
cout << "s2:␣" << s2 << endl;
s3 = s1 + '␣' + s2;
cout << "s3:␣(=␣s1␣+␣'␣'␣+␣s2):␣" << s3 << endl;
s3.replace(s3.find(s1), s1.length(), s2);
cout << "s3:␣(=␣s3.replace(s3.find(s1),␣s1.length(),\
␣s2)):␣"
     << s3 << endl;
```

...Strings: C++...

Programmausgabe

```
C:\doc\Lehre\3.Informatik_II\progs_neu\cpp-strings.exe
s1: Hallo
s2: Welt
s3 (= s1 + s2): HalloWelt
s3 + '!': HalloWelt!
name: Hans Maier
nachname (= name.sbstr(5,5)): Maier
name (name.replace(0,3,"Herbert"): Herbert Maier


Bitte zwei Worte eingeben: zwei Worte
s1: zwei
s2: Worte
s3: (= s1 + ' ' + s2): zwei Worte
s3: (= s3.replace(s3.find(s1), s1.length(), s2)): Worte Worte
```

...Strings: C++

Vergleichsoperatoren

```
string zauberspruch (/*geheim*/);
string antwort;

// Vergleich zwischen string Variablen
if (antwort == zauberspruch) {
  // Sesam oeffne dich
}
// Vergleich zwischen Variable und Zeichenliteral
else if (antwort == "hmmm...oehh" {
  // dumm gelaufen
}
```

Vektoren in C++

- C++-Strings: Speicher für Zeichenketten wird automatisch verwaltet.
- Aufgabe: Text zeilenweise in Strings einlesen: Anzahl der Zeilen vorher unbekannt
- C-Lösung: dynamische Speicherverwaltung mit Allokation von Speicher
- C++-Lösung: Benutzung von Vektoren
- Vektoren wachsen dynamisch

Bemerkung

Dies ist ein Vorgriff auf Konzepte der C++ Standard Template Library (STL).

Vektoren in C++ (Deklaration)

- Der `vector` ist in der C++-Standard-Bibliothek vorhanden, `#include <vector>`.
- `vector` ist eine Template-Klasse, d. h. eine Schablone für beliebige Typen
- Es können z. B. Vektoren von Zahlen oder Strings erzeugt werden.

Deklaration

vector<*typ*> *variable*;

Beispiel

```
vector<string> text;
```

Vektoren in C++ (Container)

- Vektoren sind Container
- Methode zum Füllen des Containers: `push_back()`, fügt ein Element an das Ende des Containers an
- Es gibt weitere Methoden, derzeit aber noch nicht interessant.
- Methode ist dem Container-Objekt mit einem Punkt (.) zugeordnet

Beispiel

```
vector<string> satz;
string wort;
cin >> wort;
satz.push_back(wort);
```

Vektoren in C++ (Zugriff)

- Lesen eines Vektor geschieht wie bei einem Feld (Array).
- Zugriff über Index, der bei 0 startet.
- Vektor besitzt eine Methode zur Berechnung der Größe, d. h. derzeitigen Anzahl enthaltener Elemente: `size()`

Beispiel

```
vector<string> satz;
// ...
cout << satz[0]; // erstes Wort
int worte = satz.size(); // Anzahl der Worte
cout << satz[worte - 1]; // letztes Wort
```

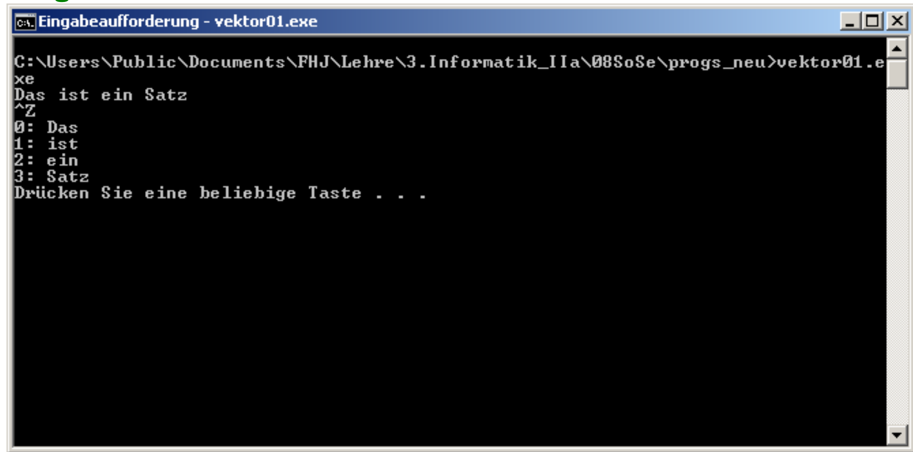
Vektoren in C++ (Beispiel)...

Beispiel Worte zählen und extrahieren

```
#include <string>
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<string> v;
  string word;
  while(cin >> word)
    v.push_back(word); // Add the line to the end
  // Add line numbers:
  for(int i = 0; i < v.size(); i++)
    cout << i << ": " << v[i] << endl;
  return EXIT_SUCCESS;
}
```

...Vektoren in C++ (Beispiel)

Programmlauf



Vektoren in C++ (Manipulation)

- Die Elemente eines Vektors können wie bei einem Feld manipuliert werden.
- Zuweisungen können nur auf existierende Elemente vorgenommen werden.

Beispiel

```
vector<int> v;
// Zuweisung auf Element
v[3] = v[3] * 15;
```

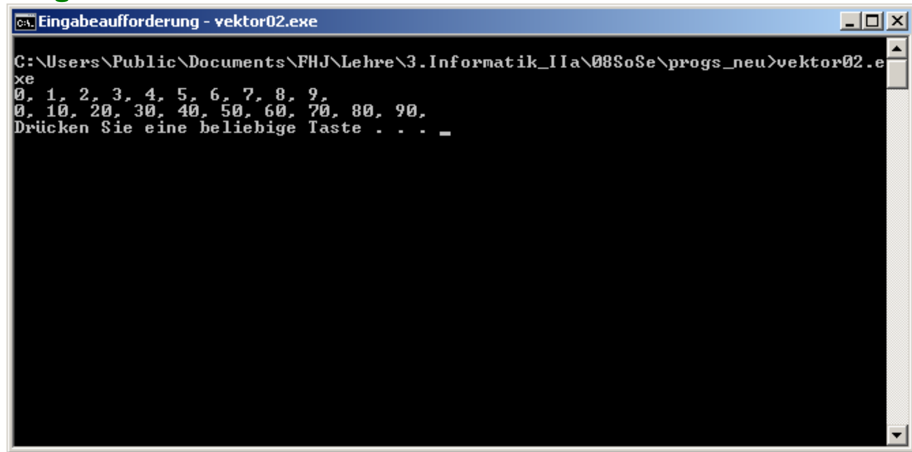
Vektoren in C++ (Beispiel)...

Beispiel Elemente mit Faktor multiplizieren

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> v;
  for(int i = 0; i < 10; i++)
    v.push_back(i);
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
  for(int i = 0; i < v.size(); i++)
    v[i] = v[i] * 10; // Assignment
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
  return EXIT_SUCCESS;
}
```

...Vektoren in C++ (Beispiel)

Programmlauf



Bemerkungen zu Datenstrukturen in C und C++

- In C++ wird weitgehend auf die Verwendung von Standard Arrays verzichtet und stattdessen der STL Vector (vector<Type>) verwendet. Dieser ist wesentlich flexibler in Bezug auf dynamisches Erzeugen, Erweitern, Initialisieren, etc. (STL ist Abkürzung für Standard Template Library)
- `typedef` kann Verständlichkeit des Variablengebrauchs erhöhen. Häufig benutzt in C++.
- Deklaration von Variablen kann an jeder Stelle im Programm erfolgen (Unterschied zu C).
- `struct`-Funktionalität erlauben auch Klassen, daher selten verwendet in C++. Mehr dazu später in der Vorlesung.

Funktionsdefinition

[*typ*] *funktionsbezeichner* (*parameterdeklarationen*)
{
    anweisungen
}

- *typ* spezifiziert den Datentyp des Rückgabewertes (Ergebnistyp)
- Soll eine Funktion keinen Rückgabewert haben, so ist *typ* mit `void` anzugeben
- Wird *typ* nicht angegeben, so ist der Datentyp des Rückgabewerts `int`

Funktionsdefinition

- *parameterdeklarationen* geben Anzahl, Position und Datentyp der Funktionsargumente an
- Beendet wird eine Funktion durch die Anweisung

  return [*ausdruck*];

  ausdruck ist Rückgabewert
- Bei `void`-Funktionen entfällt *ausdruck*

Prototypen

[*typ*] *funktionsbezeichner*(*parameterdeklarationen*);

- Eine Funktion kann deklariert werden, bevor sie definiert wird (Funktionsprototyp)
- Damit kann eine Funktion im Programm benutzt werden, ohne dass sie vorher definiert ist
- Funktionsprototypen werden zur besseren Strukturierung eines Programms eingesetzt

Beispiel Funktion

```
#include <iostream>
using namespace std;

int min (int x, int y); // Funktionsprototyp
int main ()
{ int j, k;
  cout << "Zwei␣ganze␣Zahlen?␣";
  cin >> j >> k;
  cout << min (j, k) << "␣=␣min␣(" << j << ",␣" << k << ")"
       << endl;
  return (0);
}
// Funktionsdefinition
int min (int x, int y)
{  if(x < y) { return (x); } else { return (y); }
}
```

Default-Argumente

- Bei der Funktionsdefinition und der Funktionsdeklaration können vordefinierte Argumente angegeben werden (Default-Argumente)
- Vordefinition durch Zuweisung in den *parameterdeklarationen*

  typ bezeichner = *ausdruck*
- Beim Aufruf wird dann, falls das entsprechende Argument nicht angegeben wird, der vordefinierte Wert verwendet
- Andernfalls wird der Wert des Aufruf-Arguments verwendet
- Sinnvoll, wenn ein Argument oder mehrere Argumente einer Funktion häufig mit den selben Werten belegt werden

Beispiel Default-Argumente

```
#include <iostream>
using namespace std;
int power (int n, int k = 2); // k ist als 2 vordef.
int main ()
{ int n;
  cout << "Ganze␣Zahl?␣";
  cin >> n;
  // Aufruf mit Default Arg (2)
  cout << power (n) << "␣=␣" << n << "^2" << endl;
  // Aufruf mit explizitem Arg (3)
  cout << power (n, 3) << "␣=␣" << n << "^3" << endl;
  return (0);
}
int power (int n, int k)
{ // else: rekursiver Aufruf
  if(k == 0) return(1); else return (power (n, k-1)*n);
}
```

Statische Variablen...

- Normale Variable, definiert in einer Funktion: Initialisierung bei jedem Aufruf.
- Statische Variable, definiert in einer Funktion: Initialisierung nur beim ersten Aufruf, Wert bleibt erhalten.
- Funktionen können damit ein „Gedächtnis" haben. Werte können „gemerkt" werden.
- Vorteil gegenüber globalen Variablen: Zugriff auf die gemerkten Werte ist außerhalb der Funktion nicht möglich.

...Statische Variablen...

Beispiel

```
void f (int a)
{
  while (a--) {
    static int n = 0; // Initialisierung einmal
    int x = 0; // Initialisierung bei jedem Aufruf
    cout << "n␣==␣" << n++ << ",␣x␣==␣" << x++ << endl;
  }
}
int main ()
{
  f (3);
  return (0)
}
```

. . . Statische Variablen

Ausgabe

```
n == 0, x == 0
n == 1, x == 0
n == 2, x == 0
```

Overloading...

- Funktionen können in einem Programm mehrere Bedeutungen haben (Polymorphie)
- in C++ heißt dies Overloading
- Bestimmt wird die Bedeutung durch die Signatur der Funktion.
- Die Signatur ist der Funktionskopf, d. h. Typ, Name und Folge der Parameter mit deren Typen.
  - `double mean(const int a[], int size)`
  - `double mean(const double a[], int size)`
  - `mean` hat zwei Bedeutungen.

...Overloading...

Beispiel Mittelwert von Zahlenfolgen

```
double mean(const int a[], int size)
{
  int sum = 0;
  for(int i = 0; i < size; ++i) {
    sum += a[i];  // int Arithmetik
  }
  // Typ-Konversion int->double
  return(double(sum)/double(size));
}
```

...Overloading

Beispiel Mittelwert von Zahlenfolgen

```
double mean(const double a[], int size)
{
  double sum = 0.0;
  for(int i = 0; i < size; ++i) {
    sum += a[i];  // double Arithmetik
  }
  return(sum/size);
}
```

Gültigkeitsbereiche...

- Gültigkeitsbereiche von Bezeichnern sind in C++ durch Blöcke definiert und durch { und } begrenzt
- Alle Bezeichner (z. B. von Variablen), die im Rumpf einer Funktion deklariert sind, sind außerhalb der Funktion nicht gültig (bekannt)
- Das selbe gilt für die Bezeichner der *parameterdeklarationen* des Funktionskopfs

```
{
  int a=2;                  // aeusserer Block a
  cout << a << endl;        // druckt 2
  { int a=7;                // innerer Block a
    cout << a << endl;      // druckt 7
  }
  cout << ++a << endl;      // druckt 3
}
```

...Gültigkeitsbereiche

- In C++ können, im Gegensatz zu C, Variablendeklarationen an jeder Stelle eines Blocks erscheinen
- Gültigkeitsbereich ist vom Ende der Deklaration bis zum Ende des innersten einschließenden Blocks

```
int max(int[], int size)
{
  cout << "Feldgroesse␣ist␣" << size << endl;
  int comp = c[0];                  // Deklaration von comp
  for(int i = 1; i < size; ++i)    // Deklaration von i
    if(c[i]>comp) {
      comp=c[i];
    }  // ab hier ist i nicht mehr bekannt
  return comp;
}
```

Argumentübergabe

- Call-by-value: Wert des Arguments wird kopiert, mit dieser Kopie wird gearbeitet
- Call-by-reference: Es wird mit der Variablen selbst gearbeitet
-

Argumenttypen	
Typ	**Deklaration**
Value	`function(int var)`
Constant-value	`function(const int var)`
Reference	`function(int &var)`
Constant-reference	`function(const int &var)`
Array	`function(int array[])`
Address	`function(int *var)`

- `function(int &var)` wird auch als `function(int& var)` geschrieben. Beide Schreibweisen sind semantisch gleich.

Beispiel Argumentübergabe (C-Stil)

- Call-by-reference durch Call-by-address (üblich in C)
- Umständlich, schwer verständlich

```
#include <iostream>
using namespace std;
int main ()
{
  int i = 7, j = 3;
  void ordne (int *p, int *q);
  cout << i << '\t' << j
       << endl;
  ordne (&i, &j);
  cout << i << '\t' << j
       << endl;
  return (0);
}
```

```
void ordne (int *p, int *q)
{
  int temp;

  if (*p > *q) {
    temp = *p;
    *p = *q;
    *q = temp;
  }
}
```

Beispiel Argumentübergabe (C++-Stil)

- Call-by-reference direkt (in C++ üblich)
- Leicht verständlich, leicht benutzbar

```
#include <iostream>
using namespace std;
int main ()
{
  int i = 7, j = 3;
  void ordne (int &p, int &q);
  cout << i << '\t' << j
       << endl;
  ordne (i, j);
  cout << i << '\t' << j
       << endl;
  return (0);
}
```

```
void ordne (int &p, int &q)
{
  int temp;

  if (p > q) {
    temp = p;
    p = q;
    q = temp;
  }
}
```

Call-by-value vs. Call-by-reference

Falsch:

```
#include <iostream>
using namespace std;
void eingabe(int i)
{ cout << "Zahl?␣"; cin >> i;
}
int main ()
{ int x = 0;
  eingabe(x);
  cout << x << endl;
  return (0);
}
```

Eingabe und Ausgabe:

```
Zahl? 2
0
```

Richtig:

```
#include <iostream>
using namespace std;
void eingabe(int &i)
{ cout << "Zahl?␣"; cin >> i;
}
int main ()
{ int x = 0;
  eingabe(x);
  cout << x << endl;
  return (0);
}
```

Eingabe und Ausgabe:

```
Zahl? 2
2
```

Constant-value, Constant-reference

- `const` bedeutet: In der Funktion kann der Wert des Parametern nicht geändert werden.
- Speziell Constant-reference: Wert des Objekts, auf den die Referenz verweist, kann in der Funktion nicht geändert werden.
- Bei großen Datentypen sinnvoll, da Speicherplatz gespart wird.

```
void f (const Gross& arg)
{ // Wert von arg kann in Funktion
  // nicht geaendert werden
}
```

- Konvention

```
void g (Gross& arg); // g() wird arg aendern
```

Rekursion

- Als Rekursion bezeichnet man den Aufruf oder die Definition einer Funktion durch sich selbst.
- Beispiel Fakultät: $0! := 1; n! := n * (n-1)!$, falls $n > 0$

```
int fak (int n) { return (n<2) ? 1 : n*fak (n-1); }
```

- Beispiel Fibonacci-Zahlen:
  $f(0) := 0; f(1) := 1; f(n) := f(n-1) + f(n-2)$, falls $n > 1$

```
int f (int n) { return (n<2) ? n : f (n-1) + f (n-2); }
```

- Implementierung von f ist ungünstig. Warum?

Beispiel Rekursion

Programmausgabe

```
C:\doc\Lehre\3.Informatik_II\Uebung\p
Bitte eine Zahl eingeben: 15
Das Ergebnis ist 987
Anzahl rekursiver Aufrufe: 1219_
```

Operatoren

typ operator *op*(*parameterdeklarationen*) {
    anweisungen
}

- In C++ können auch Operatoren Funktionsbezeichner sein.
- Operatorfunktionen werden für Overloading von Zuweisungsoperatoren, relationalen und arithmetischen Operatoren etc. verwendet.
- So können für einen neu definierten Datentyp entsprechende Operatoren definiert werden. Das erhöht die Lesbarkeit des Programmtextes.
- Alle C++-Operatoren können so für entsprechende Datentypen neu definiert werden.

Beispiel Operatoren...

complex.h

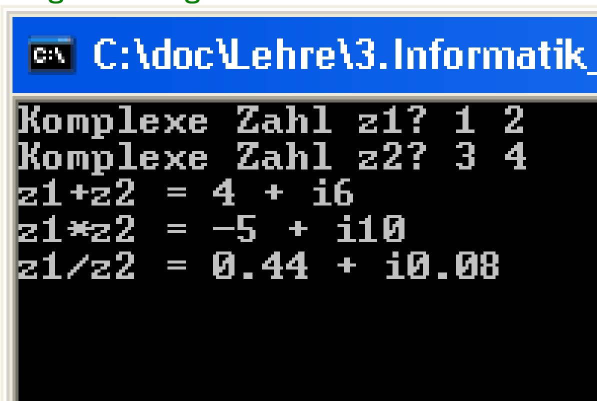
```
// Datentyp complex
struct complex {
  float re;
  float im;
};

// Operatorfunktion +, Addition komplexer Zahlen
complex operator +(complex &zahl1, complex &zahl2)
{
  complex erg;
  erg.re = zahl1.re + zahl2.re;
  erg.im = zahl1.im + zahl2.im;
  return (erg);
}
```

...Beispiel Operatoren...

```
#include <iostream>
#include "complex.h"
using namespace std;
int main ()
{ complex z1, z2, z3;
  cout << "Komplexe␣Zahl␣z1?␣";
  cin >> z1.re >> z1.im;
  cout << "Komplexe␣Zahl␣z2?␣";
  cin >> z2.re >> z2.im;
  z3 = z1 + z2;
  cout << "z1+z2␣=␣" << z3.re << "␣+␣i" << z3.im << '\n';
  z3 = z1 * z2;
  cout << "z1*z2␣=␣" << z3.re << "␣+␣i" << z3.im << '\n';
  z3 = z1 / z2;
  cout << "z1/z2␣=␣" << z3.re << "␣+␣i" << z3.im << '\n';
  return (0);
}
```

. . .Beispiel Operatoren

Programmausgabe



Datentypen, Funktionen und Operatoren

- Für Zeichenketten gibt es einen eigenen Datentyp string mit zugehörigen Operationen.
- Reihungen von beliebigen Elementen können mit dem parametrischen Datentyp vector realisiert werden.
- Funktionen haben einen Typ und Parameter, die mit Default-Werten belegt sein können.
- Statische Variablen innnerhalb von Funktionen werden nur einmal initialisiert.
- In C++ können Funktionen mehrere Bedeutungen haben (Overloading).
- In C++ können Argumente an Funktionen als Call-by-Value und Call-by-reference übergeben werden.
- Wenn eine Funktion sich selbst aufruft oder durch sich selbst definiert ist, wird sie als rekursive Funktion bezeichnet.
- In C++ können auch Operstoren Funktionsbezeichner sein und selbst definiert werden.