Rechtlicher Hinweis

Diese Präsentation ist urheberrechtlich geschützt und darf nur im Rahmen von Lehrveranstaltungen der Friedrich-Schiller-Universität Jena verwendet werden. Eine Nutzung durch Verbreitung oder Veröffentlichung dieses Materials - auch in Auszügen - ist strengstens untersagt und wird die Geltendmachung von Unterlassungs- und Schadenersatzansprüchen durch die Friedrich-Schiller-Universität Jena zur Folge haben.

Legal notice

These slides are protected by copyright and may only be used as part of courses at the Friedrich Schiller University Jena. Any use through the dissemination or publication of this material - even in extracts - is strictly prohibited and will result in the assertion of injunctive relief and claims for compensation by the Friedrich Schiller University Jena.

Informatik I (B.Sc. Physik)

Variablen und Datentypen

Dr. Paul Bodesheim
(Paul.Bodesheim@uni-jena.de)

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Fakultät für Mathematik und Informatik
Lehrstuhl für Digitale Bildverarbeitung

SoSe 2020

Inhalt

Inhalt

Ausgangspunkt: “Hallo Welt!”

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hallo Welt!" << endl;

    return 0;
}
```

Beispiel erweitern (Nutzereingabe)

来自计算机的个人问候
要求提供姓名
提醒：cout使输出到控制台
用于从控制台读取的cin
箭头转：cin >>（与cout <<相反）
放在哪里？
更精确地存储在某处：写入内存
）变量

- Persönliche Begrüßung durch den Rechner
- Erfordert erfragen des Namens
- Erinnerung: *cout* ermöglicht Ausgabe auf Konsole

- **cin** für Einlesen von Konsole
- Pfeile drehen sich um: cin >> (im Gegensatz zu cout <<)
- Wohin mit der Eingabe?

- Irgendwo ablegen, genauer: in den Speicher schreiben

  $\Rightarrow$ **Variablen**

Inhalt

Was sind Variablen?

- In der Mathematik: "Platzhalter" für Zahlen / Werte
- Beim Programmieren: Variablen stellen Platz im Speicher dar
- Speicherplatz braucht Namen, um ihn ansprechen / adressieren zu können

  $\Rightarrow$ *Variablenname*

- Aber wieviel Speicherplatz? Wieviele Bits / Bytes?
- Das übernimmt der Compiler und das System
- Der Programmierer muss nur feslegen, was dort abgelegt werden soll

  $\Rightarrow$ *Typ / Datentyp*

在数学中：" \占位符 "用于数字/值
编程时：变量表示内存中的空间。
内存空间需要名称才能寻址/寻址。
但是有多少空间？ 多少位/字节？
这是由编译器和系统完成的
程序员只需指定应该在此处存储的内容）类型/数据类型

Ein Beispiel für Variable und Typ

- Namen des Nutzers erfragen
- Name = Text = Folge von Zeichen
- In der Informatik: Zeichenkette, englisch: string

  ⇒ Name des Datentyps für Zeichenketten in C++

询问用户名
名称=文字=字符序列
在计算机科学中：字符串，英语：字符串
）C ++中字符串的数据类型名称
创建变量：字符串名称；
第一个类型（类型名称），然后是变量（变量名称）
类型必须存在（选择）
可以根据某些规则自行设置变量名称

- Variable anlegen: **string** name;
- Erst Typ (Name des Typs), dann Variable (Variablenname)
- Typ muss es geben (auswählen)
- Variablenname kann selber festgelegt werden, nach bestimmten Regeln

Inhalt

Bezeichner (identifier)

- Der Name einer Variable ist ein Bezeichner
- In C++ verschiedene Regeln für Bezeichner:
  - Mit Buchstaben (groß oder klein) beginnen (erstes Zeichen)
  - Danach Ziffern und Buchstaben in beliebiger Reihenfolge
  - Keine Leerzeichen, keine Sonderzeichen erlaubt (Ausnahme: Unterstrich)
- Definition für das Anlegen von Variablen:
  `<type> <identifier> {, <identifier>};`
- Anlegen mehrerer Variablen des gleichen Typs in einer Anweisung erlaubt
- Beispiele:
  - string name;
  - string s1,s2;
  - string r2_d2;

变量的名称是一个标识符
C ++中标识符的不同规则：
以字母（大或小）开头（第一个字符）
然后以任意顺序排列数字和字母
没有空格，不允许使用特殊字符（下划线除外）
创建变量的定义：
`<type> <identifier> {, <identifier>};`
可以在一条指令中创建相同类型的多个变量

Empfehlungen für Variablennamen

- Sorgfältige Namensauswahl: aussagekräftige Variablennamen!
  $\Rightarrow$ Code besser lesbar, leichter verständlich, wiederverwendbar
- **Sprechende Notation**:
  Namen drücken aus, was in der Variable gespeichert wird
- Beispiele: name, alter
- Bei mehreren Wörtern: **CamelCase-Schreibweise**
  - Wörter ohne Leerzeichen aneinanderhängen (Leerzeichen sind in Bezeichnern ohnehin nicht erlaubt)
  - jedes neue Wort beginnt mit Großbuchstabe, sonst Kleinbuchstaben
  - Beispiele: summeDistanzen, sumOfDistances
- Alternativ: Unterstrich zwischen den Wörtern, z.B.: sum_of_distances
- Meine Empfehlung: Notation mit Unterstrich für Funktionsnamen (später), CamelCase für Variablen

Inhalt

Zeichenketten einlesen

- Um Zeichenketten im Programm verwenden zu können:
  #include <string>
- string name;
- Anlegen einer Variable = Speicher zur Verfügung stellen
- Bei vielen Typen: automatische Initialisierung
  ⇒ Bei string ist das Leerstring: ""
  ⇒ Leerstring ≠ Leerzeichen ("" ≠ " ")
- Namen auf Konsole eingeben: **cin >> name;**
  ⇒ Eingabe einer Zeichenkette bis zum Betätigen der Enter-Taste
  ⇒ Eingabe wird in Variable *name* gespeichert

Erweitertes “Hallo Welt!”

- Personalisierte Ausgabe möglich:
  cout << ’’Hallo ’’ << name << endl;
- Verwendung der Variable:
  Variable repräsentiert Wert, der zur Zeit auf dem Speicherplatz steht

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string name; // Variable fuer Name anlegen
    cout << "Wie lautet dein Name? ";
    cin >> name;
    cout << "Hallo " << name << endl;

    return 0;
}
```

Zusammenfassung: Variablen

- Stellen Speicherplatz mit Namen dar (Name ist Bezeichner)
- Haben einen Typ, der über Speicherplatzgröße entscheidet (und zulässige Operationen definiert)
- Repräsentieren den Wert, der aktuell im Speicher abgelegt ist
- Müssen angelegt / definiert werden, bevor sie verwendet werden können

Inhalt

Verschiedene Datentypen

- Im Beispiel: weitere Angaben des Nutzers erfragen
- Name ist Zeichenkette, Zahlen für Alter, Größe, Gewicht
- Unterschiede bei Zahlen (numerische Werte):
  - Ganze Zahlen (Alter)
  - Rationale Zahlen (Größe)
  - Reelle Zahlen
- Logische Werte: wahr / falsch
- Später: Eigene / komplexe / zusammengesetzte Datentypen
- Standard-Datentypen für numerische Werte sind immer vorhanden und müssen nicht über eine Bibliothek mit include eingebunden werden, ebenso logische Werte (im Gegensatz zu string)

Inhalt

Ganze Zahlen

- Ganze Zahlen für Anzahl (z.B. Anzahl Geschwister)
- Englische Bezeichnung für ganze Zahl: integer
- Datentyp: **int** (als Abkürzung für integer)
- In der Mathematik: unendlicher Wertebereich
- Beim Programmieren: endlicher Wertebereich, da endlicher Speicher (endliche Anzahl Bits)

  $\Rightarrow$ verschiedene Typen ganzer Zahlen mit verschieden großen Wertebereichen
- Wertebereiche der Typen systemabhängig

Typen für ganze Zahlen

Typ	Speicherbedarf nach C++11	Speicherbedarf (typisch, mein Rechner)	Entsprechender Wertebereich (typisch, mein Rechner)
int	$\geq$ 16 Bits	32 Bits	$-2^{31} .. 2^{31} - 1$ $\approx -2 \cdot 10^9 .. \approx 2 \cdot 10^9$
short short int	$\geq$ 16 Bits	16 Bits	$-2^{15} .. 2^{15} - 1$ $-32768 .. 32767$
long long int	$\geq$ 32 Bits	64 Bits	$-2^{63} .. 2^{63} - 1$ $\approx -9 \cdot 10^{18} .. \approx 9 \cdot 10^{18}$
long long int	$\geq$ 64 Bits	64 Bits	siehe long int

- Für ganze Zahlen üblicherweise: **int**
- Vergleiche: Zahlendarstellung im Rechner (später)

Ganze Zahlen: vorzeichenbehaftet oder vorzeichenlos

- Zu jeder Speichergröße auch möglich: nur positive Zahlen (**unsigned**, vorzeichenlos)
- Bei vorzeichenbehafteten Typen kann **signed** explizit angegeben werden
  $\Rightarrow$ kann weggelassen werden, weil vorzeichenbehaftet ist Standard
- Vorteil von unsigned (vorzeichenlos): größerer Wertebereich (nur positiv)
  $\Rightarrow$ bessere Speicherausnutzung wenn man nur positive Werte garantieren kann, z.B. Anzahlen / Zählungen repräsentieren
- Oft Verzicht auf unsigned, da Gewinn nur 1 Bit und Wertebereich auf bei vorzeichenbehafteten Typen ausreichend groß ist für die meisten Anwendungen
- Außerdem: Verwendung von unsigned sehr fehleranfällig

Erweiterte Übersicht: Typen für ganze Zahlen

Typ	Speicherbedarf (typisch, mein Rechner)	Entsprechender Wertebereich (typisch, mein Rechner)
int signed int	32 Bits	-2^{31} .. $2^{31}-1$ $\approx -2.1 \cdot 10^9$ .. $\approx 2.1 \cdot 10^9$
unsigned int	32 Bits	0 .. $2^{32}-1$ 0 .. $\approx 4.2 \cdot 10^9$
short int signed short int	16 Bits	-2^{15} .. $2^{15}-1$ -32768 .. 32767
unsigned short int	16 Bits	0 .. $2^{16}-1$ 0 .. 65535
long int signed long int	64 Bits	-2^{63} .. $2^{63}-1$ $\approx -9.2 \cdot 10^{18}$ .. $\approx 9.2 \cdot 10^{18}$
unsigned long int	64 Bits	0 .. $2^{64}-1$ 0 .. $\approx 1.8 \cdot 10^{19}$
long long int signed long long int	64 Bits	siehe long int
unsigned long long int	64 Bits	siehe unsigned long int

Besonderer Datentyp: char

- Datentyp für einzelne Zeichen (character):
  - Ziffern als Zeichen: '0'-'9'
  - Buchstaben: 'a'-'z','A'-'Z'
  - Sonderzeichen: '.' ',' '?' '!' '-' '/' ...
  - ...

  $\Rightarrow$ Später mehr dazu...

- char auch als numerischer Datentyp verwendbar
- Speicherbedarf: 8 Bit (1 Byte)

- Wertebereich unsigned char: $0 .. 2^8 - 1$ $(0 .. 255)$
- Wertebereich signed char: $-2^7 .. 2^7 - 1$ $(-128 .. 127)$

- Wenn char als numerischer Typ verwendet werden soll, dann immer als signed oder unsigned spezifizieren

Probleme durch begrenzte Wertebereiche

- Differenz zweier ganzer Zahlen eines unsigned-Typs ist ebenfalls vom unsigned-Typ

  ⇒ Ergebnis ist immer positiv, egal welche Ausgangszahl bei der Differenzbildung größer war

  ⇒ **Unterlauf (underflow)**, wird von C++ nicht gemeldet

  ⇒ Es wird mit falschem Ergebnis weitergearbeitet

- Gegenstück: **Überlauf (overflow)**

  ⇒ Überschreiten der oberen Grenze des Wertebereiches, z.B. durch Summe großer Zahlen

  ⇒ Beispiel: Statistiken für Farbwerte eines Bildes (Summe aller Pixelwerte oder deren Quadrate)

- Konvertierung zwischen signed und unsigned möglich, aber fehlerbehaftet

  ⇒ Negative Zahlen werden zu großen positiven Zahlen und umgekehrt

  ⇒ Keine Konvertierung im technischen Sinne, sondern einfach andere Interpretation des Speichers (der Bits)

  ⇒ Wird nicht vom Compiler gemeldet

Ermitteln des Wertebereiches

- In C++ über templates
- An dieser Stelle: keine Erläuterung, was templates sind

- Benötigen zusätzliche Funktionalitäten:
- **#include** <**limits**>

- cout << numeric_limits<**long int**>::min() << " .. ";
- cout << numeric_limits<**long int**>::max() << endl;

Inhalt

Gleitkommazahlen

- Bezeichnung für Darstellung rationaler Zahlen in der Informatik
- Bildlich: verschiebbares Komma

  $\Rightarrow$ Nur rationale Zahlen, da Speicher endlich

  $\Rightarrow$ Reelle Zahlen lediglich angenähert darstellbar
- Unterscheidung nach Zahlenbereich und Genauigkeit:
  - Unterschiedlich große Zahlenbereiche (Unter- und Obergrenze)
  - Unterschiedliche Genauigkeiten: nicht alle Zwischenwerte lassen sich darstellen und müssen angenähert werden (selbst wenn es rationale Zahlen sind)

Typen für Gleitkommazahlen

Typ	Speicherbedarf	Entsprechender Wertebereich
float	32	$\approx -3.4 \cdot 10^{38}$ .. $\approx 3.4 \cdot 10^{38}$
double	64	$\approx -1.8 \cdot 10^{308}$ .. $\approx 1.8 \cdot 10^{308}$
long double	80	$\approx -1.2 \cdot 10^{4932}$ .. $\approx 1.2 \cdot 10^{4932}$

- Üblicherweise: **double**
- Wichtig: Komma wird als Punkt dargestellt (0.5 oder 3.1415)
- Repräsentation von gebrochenen Zahlen in Exponentialschreibweise mit Basis 2

  $\Rightarrow$ Vergleiche: Zahlendarstellung im Rechner (später)

Erweiterungen zu “Hallo Welt!”

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string name; // Variable fuer Name anlegen
    cout << "Wie lautet dein Name? ";
    cin >> name;
    cout << "Hallo " << name << endl;

    int alter;
    cout << "Wie alt bist du? ";
    cin >> alter;

    double groesse;
    cout << "Wie gross bist du? ";
    cin >> groesse;

    // Weitere Verwendung von alter und groesse

    return 0;
}
```

Inhalt

Wahrheitswerte

- Überprüfen von Aussagen wie "Ist $x > 5$?"
- Entweder richtig oder falsch (ja oder nein)

- Datentyp: **bool**
- Gültige Werte: **true** und **false**

- Repräsentiert ein Bit
- Auch als Schalter oder flag bezeichnet

- Später wichtig bei logischen Ausdrücken und Kontrollstrukturen

Inhalt

Beispiel: einfache Dreiecksberechnungen

```
#include <iostream>

using namespace std;

int main()
{
    double a,b,c; // Variablen anlegen

    cout << "Seitenlaengen des Dreiecks eingeben!" << endl;

    cout << "a: ";
    cin >> a;

    cout << "b: ";
    cin >> b;

    cout << "c: ";
    cin >> c;

    cout << "Der Umfang ist " << a + b + c << endl;

    return 0;
}
```

Was passiert im Beispiel?

- Variablen für Seitenlängen anlegen
- Nutzer zu Eingabe auffordern
- Seitenlängen nach und nach eingeben lassen und Werte in Variablen speichern (hier noch keine Überprüfung, ob wirklich numerische Werte eingegeben wurden)
- Umfangsberechnung direkt im Ausgabebefehl: Summe der 3 Werte

- Berechnungen ähnlich wie in der Mathematik
- Ausdruck ist Rechenvorschrift, Summe (+) als Operator später nochmal (Abarbeitung von links nach rechts, Punkt- vor Strichrechnung, ...)

- Was ist, wenn der Umfang wiederverwendet werden soll? (z.B. für Flächenberechnung)
  - Neuberechnung? Nein!
  - Wert merken: Zuweisung zu einer Variablen

Zuweisungen von Werten

```
double umfang;
umfang = a + b + c;
```

- Zuweisung: (ggf. berechneter) Wert eines Ausdrucks in einer Variable speichern
- Zuweisung ist in C++ auch Operator

  (Ausdrücke und Operatoren später nochmal etwas genauer)
- **Zuweisungsoperator =**

- Zuweisung $\neq$ Gleichung in der Mathematik
- Zuweisung: Variable auf der linken Seite erhält Wert des Ausdrucks auf der rechten Seite
- Definition: <lvalue> = <rvalue>;
- Zuerst wird <rvalue> bestimmt und dann der linken Seite (typischerweise einer Variablen) zugewiesen

Beispiel: Flächenberechnung

```
// Programmausschnitt, vorher: include, namespace
// Dann Beginn der Hauptfunktion: int main()
// Danach Seitenlaengen einlesen wie im Beispiel zuvor

double umfang;

umfang = a + b + c; // Umfang berechnen

cout << "Der Umfang ist " << umfang << endl; // Umfang ausgeben


double s = umfang/2; // Hilfsvariable

double flaeche = sqrt(s * (s-a) * (s-b) * (s-c));

cout << "Die Flaeche ist " << flaeche << endl; // Flaeche ausgeben

// Am Ende: "return 0;" und "}" fuer Abschluss der main-Methode
```

Erläuterungen zum Beispiel

- Umfang $u = a + b + c$
- Fläche nach Satz von Heron: $A = \sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)}$ mit $s = \frac{u}{2}$

- Rechenarten (Operatoren):
  Addition (+), Subtraktion (-), Multiplikation (∗), Division (/)
- Punkt- vor Strichrechnung, sonst runde Klammern

- Wurzel als Funktion: **sqrt** (Argument in runden Klammern)

  ⇒ Verwendung erfordert **#include** <cmath>

Variablen initialisieren

- (Ausgangs-) Wert beim Anlegen der Variablen zuweisen
- Beispiel: **double** umfang = 0.0;

- **Wichtig: Variablen immer mit Standardwerten (z.B. numerische Werte mit 0, string mit "") initialisieren**
- Sonst: Gefahr der Verwendung von unbestimmten Werten

```
double d; // Wert unbestimmt
cout << d; // Ausgabe kann zufaelliger Wert sein
// System nimmt eventuell Standardwert, das ist aber nicht
    sicher!

// Weitere Anweisungen

d = d + 1;
cout << d;
```

Formen der Initialisierung

```
double d = 1.5; // Zuweisungsoperator =

int a(5); // runde Klammern

int b{-7}; // geschweifte Klammern, nach C++11 Standard

double c = {10}; // Zuweisungsoperator und geschweifte Klammern
```

- Compiler liefert Warnung für Initialisierung von b, wenn nicht mit der Option "-std=c++11" übersetzt wird
- Programm wird trotzdem korrekt gebaut und ist lauffähig (tut auch, was es soll)

Initialisierung $\neq$ Zuweisung

```
int a; // Variable anlegen
a = 10; // Wert zuweisen
```

```
// Anlegen mit Initialisierung
int a = 10;
```

- Verwechselungsgefahr: Zuweisung oder Initialisierung mit $=$
- Beide Varianten von der Auswirkung her identisch, eventuell sogar gleicher Maschinencode

- Wichtig für komplexere Typen (später): Wirkung gleich, aber Ausführung unterschiedlich
- Anlegen und Zuweisung können länger dauern als Anlegen mit Initialisierung
- Besser: Anlegen mit Initialisierung ist nie langsamer als Anlegen und Zuweisung
- Daher Empfehlung: Variablen immer mit (sinnvoller) Initialisierung anlegen

Überblick: Initialisierung

- **Bei fehlender Initialisierung ist der Wert einer Variablen unbestimmt!**
- Besser Variable initialisieren und Wert überschreiben, statt Gefahr zu laufen einen unbestimmten Wert zu verwenden
- Aus Performanzgründen: wenn möglich Variable beim anlegen initialisieren, statt erst anlegen und anschließend Wertzuweisung
- Unterschiedliche Formen der Initialisierung, mit $=$ oder () am gebräuchlichsten

Inhalt

Lebensdauer und Sichtbarkeit von Variablen

- Gültigkeit: Variablen sind gültig ab ihrer Definition (Anlegen) bis zum Ende des Blocks, indem sie definiert wurden

- **Globale Variablen:**
  - Definition außerhalb aller Blöcke
  - Von überall zugreifbar
  - Lebenszeit ist Laufzeit des gesamten Programms

- **Lokale Variablen:**
  - Nur in dem Block erreichbar, indem sie definiert wurden (Block ist Funktion oder Teil einer Funktion)
  - Zerstörung am Ende des Blocks
  - Bei wiederholtem Durchlaufen des Blocks: Variable wird neu angelegt (alter Wert geht verloren)

Beispiel: Lebensdauer und Sichtbarkeit

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

double p = 3.1415; // nur fuer Konstanten zu empfehlen

int main()
{
    cout << p << endl; // 3.1415
    double p=1.25;
    {
        int p = 10;
        cout << p << endl; // 10
    }
    cout << p << endl; // 1.25

    return 0;
}
```

- Sichtbarkeit: lokales Überdecken in Blöcken möglich (gleicher Name, aber unterschiedlicher Wert, ggf. sogar verschiedener Typ)
- Mehrfaches Anlegen mit gleichem Namen in gleichem Block geht nicht!

  $\Rightarrow$ Eindeutigkeit innerhalb eines Blocks

Inhalt

Variablen als Konstanten

- In manchen Anwendungen wird ein fester (unveränderlicher) Wert mehrmals verwendet
- Es soll sichergestellt werden, dass dieser Wert sich im Laufe des Programms nicht verändert
- Konstanten:
  - Anlegen wie Variablen, nur mit zusätzlichem Attribut (auch lokal möglich)
  - Schlüsselwort: **const**
  - Beispiel: **const double** pi=3.1415;
  - Wert darf im Programm nicht verändert werden, sonst Fehlermeldung durch Compiler beim Übersetzen des Programms
    $\Rightarrow$ Konstanten dürfen nie auf der linken Seite einer Zuweisung stehen
  - Sonst: Verwendung wie Variablen (nur lesender Zugriff, nicht schreibend)
  - Compiler berücksichtigt Konstantheit für Optimierung des Codes

Inhalt

Empfehlungen

- Globale Variablen vermeiden! (höchstens für Konstanten)
  $\Rightarrow$ In der Regel einfach umzusetzen
- Variablen so lokal wie möglich definieren
  $\Rightarrow$ Nur in den Blöcken, wo sie gebraucht werden
- Unveränderliche (feste) Werte als Konstanten definieren (**const**)

Zwei Möglichkeiten für die Definition von Variablen

1. Alle Variablen eines Blocks am Anfang des Blocks definieren und initialisieren
   - Vorteil: alle Definitionen an gleicher Stelle (übersichtlich, leicht auffindbar)
   - Nachteil: Definition und Initialisierungswert ggf. weit weg von erster Verwendung
2. Variablen an der Stelle im Block definieren und initialisieren, wo sie gebraucht werden
   - Vorteil: initialer Wert unmittelbar vor Verwendung zugewiesen
   - Nachteil: keine Übersicht über alle lokalen Variablen eines Blocks

Beispiel

```
#include <iostream>

using namespace std;

int main()
{
    int a=2;
    int b=4;
    double c=0.5;
    double d=0.125;

    a = a+b;
    b = a*a;

    c = c*(a + b);
    cout << c << endl;

    c = a*c + b*d;
    cout << c*d << endl;

    return 0;
}
```

```
#include <iostream>

using namespace std;

int main()
{
    int a=2;
    int b=4;

    a = a+b;
    b = a*a;

    double c=0.5;
    c = c*(a + b);
    cout << c << endl;

    double d=0.125;
    c = a*c + b*d;
    cout << c*d << endl;

    return 0;
}
```

Gibt es Fragen?

(Es gibt keine dummen Fragen!)