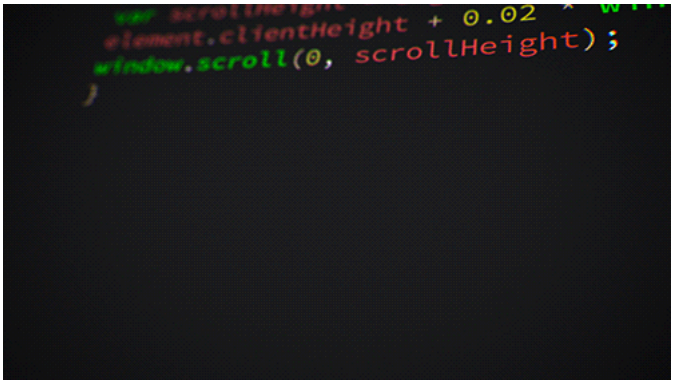
Grundlagen der Sprache C

Parameter	Kursinformationen
Veranstaltung:	`Einführung in das wissenschaftliche Programmieren`
Semester	`Wintersemester 2022/23`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Ein- und Ausgabe / Variablen`
Link auf Repository:	https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/01_EingabeAusgabeDatentypen.md
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf



Fragen an die heutige Veranstaltung ...

- Durch welche Parameter ist eine Variable definiert?
- Erklären Sie die Begriffe Deklaration, Definition und Initialisierung im Hinblick auf eine Variable!
- Worin unterscheidet sich die Zahldarstellung von ganzen Zahlen (`int`) von den Gleitkommazahlen (`float`).
- Welche Datentypen kennt die Sprache C++?
- Erläutern Sie für `char` und `int` welche maximalen und minimalen Zahlenwerte sich damit angeben lassen.
- Ist `printf` ein Schlüsselwort der Programmiersprache C++?
- Welche Beschränkung hat `getchar`

Vorwarnung: Man kann Variablen nicht ohne Ausgaben und Ausgaben nicht ohne Variablen erklären. Deshalb wird im folgenden immer wieder auf einzelne Aspekte vorgegriffen. Nach der Vorlesung sollte sich dann aber ein Gesamtbild ergeben.

Variablen

Lassen sie uns den Rechner als Rechner benutzen ... und die Lösungen einer quadratischen Gleichung bestimmen:

$$y = 3x^2 + 4x + 8$$

QuadraticEquation.cpp

```cpp
#include<iostream>

int main() {
  // Variante 1 - ganz schlecht
  std::cout <<"f("<<x<<") = "<<3*5*5 + 4*5 + 8<<" \n";

  // Variante 2 - besser
  int x = 9;
  std::cout <<"f("<<x<<") = "<<3*x*x + 4*x + 8<<" \n";
  return 0;
}
```

```
CodeRunner is not defined
```

Unbefriedigende Lösung, jede neue Berechnung muss in den Source-Code integriert und dieser dann kompiliert werden. Ein Taschenrechner wäre die bessere Lösung!

Ein Programm manipuliert Daten, die in Variablen organisiert werden.

Eine Variable ist ein **abstrakter Behälter** für Inhalte, welche im Verlauf eines Rechenprozesses benutzt werden. Im Normalfall wird eine Variable im Quelltext durch einen Namen bezeichnet, der die Adresse im Speicher repräsentiert. Alle Variablen müssen vor Gebrauch vereinbart werden.

Kennzeichen einer Variable:

1. Name
2. Datentyp
3. Wert
4. Adresse
5. Gültigkeitraum

Mit `const` kann bei einer Vereinbarung der Variable festgelegt werden, dass ihr Wert sich nicht ändert.

```cpp
const double e = 2.71828182845905;
```

Ein weiterer Typqualifikator ist `volatile`. Er gibt an, dass der Wert der Variable sich jederzeit z. B. durch andere Prozesse ändern kann.

Zulässige Variablennamen

Der Name kann Zeichen, Ziffern und den Unterstrich enthalten. Dabei ist zu beachten:

- Das erste Zeichen muss ein Buchstabe sein, der Unterstrich ist auch möglich.
- C++ betrachte Groß- und Kleinschreibung - `Zahl` und `zahl` sind also unterschiedliche Variablennamen.
- Schlüsselworte (`class`, `for`, `return`, etc.) sind als Namen unzulässig.

Name	Zulässigkeit
`gcc`	erlaubt
`a234a_xwd3`	erlaubt
`speed_m_per_s`	erlaubt
`double`	nicht zulässig (Schlüsselwort)
`robot.speed`	nicht zulässig (`.` im Namen)
`3thName`	nicht zulässig (Ziffer als erstes Zeichen)
`x y`	nicht zulässig (Leerzeichen im Variablennamen)

QuadraticEquation.cpp

```cpp
#include<iostream>

int main() {
  int x = 5;
  std::cout<<"Unsere Variable hat den Wert "<<x<<" \n";
  return 0;
}
```

```
CodeRunner is not defined
```

> Vergeben Sie die Variablennamen mit Sorgfalt. Für jemanden der Ihren Code liest, sind diese wichtige Informationsquellen! Link

Neben der Namensgebung selbst unterstützt auch eine entsprechende Notationen die Lesbarkeit. In Programmen sollte ein Format konsistent verwendet werden.

Bezeichnung	**denkbare Variablennamen**
CamelCase (upperCamel)	`YouLikeCamelCase`, `HumanDetectionSuccessfull`
(lowerCamel)	`youLikeCamelCase`, `humanDetectionSuccessfull`
underscores	`I_hate_Camel_Case`, `human_detection_successfull`

In der Vergangenheit wurden die Konventionen (zum Beispiel durch Microsoft "Ungarische Notation") verpflichtend durchgesetzt. Heute dienen eher die generellen Richtlinien des Clean Code in Bezug auf die Namensgebung.

Datentypen

Welche Informationen lassen sich mit Blick auf einen Speicherauszug im Hinblick auf die Daten extrahieren?

Adresse	Speicherinhalt
	binär
0010	0000 1100
0011	1111 1101
0012	0001 0000
0013	1000 0000

Adresse	Speicherinhalt	Zahlenwert
		(Byte)
0010	0000 1100	12
0011	1111 1101	253 (-3)
0012	0001 0000	16
0013	1000 0000	128 (-128)

Adresse	Speicherinhalt	Zahlenwert	Zahlenwert	Zahlenwert
		(Byte)	(2 Byte)	(4 Byte)
0010	0000 1100	12		
0011	1111 1101	253 (-3)	3325	
0012	0001 0000	16		
0013	1000 0000	128 (-128)	4224	217911424

Der dargestellte Speicherauszug kann aber auch eine Kommazahl (Floating Point) umfassen und repräsentiert dann den Wert `3.8990753E-31`

Folglich bedarf es eines expliziten Wissens um den Charakter der Zahl, um eine korrekte Interpretation zu ermöglichen. Dabei erfolgt die Einteilung nach:

- Wertebereichen (größte und kleinste Zahl)
- ggf. vorzeichenbehaftet Zahlen
- ggf. gebrochene Werte

Ganze Zahlen, `char` und `bool`

Ganzzahlen sind Zahlen ohne Nachkommastellen mit und ohne Vorzeichen. In C/C++ gibt es folgende Typen für Ganzzahlen:

Schlüsselwort	Benutzung	Mindestgröße
`char`	1 Byte bzw. 1 Zeichen	1 Byte (min/max)
`short int`	Ganzahl (ggf. mit Vorzeichen)	2 Byte
`int`	Ganzahl (ggf. mit Vorzeichen)	"natürliche Größe"
`long int`	Ganzahl (ggf. mit Vorzeichen)	
`long long int`	Ganzahl (ggf. mit Vorzeichen)	
`bool`	boolsche Variable	1 Byte

```
signed char <= short <= int <= long <= long long
```

Gängige Zuschnitte für `char` oder `int`

Schlüsselwort	Wertebereich
`signed char`	-128 bis 127
`char`	0 bis 255 (0xFF)
`signed int`	-32768 bis 32767
`int`	65536 (0xFFFF)

Wenn die Typ-Spezifizierer (`long` oder `short`) vorhanden sind kann auf die `int` Typangabe verzichtet werden.

```
short int a; // entspricht short a;
long int b;  // äquivalent zu long b;
```

Standardmäßig wird von vorzeichenbehafteten Zahlenwerten ausgegangen. Somit wird das Schlüsselwort `signed` eigentliche nicht benötigt

```
int a;  //  signed int a;
unsigned long long int b;
```

Sonderfall `char`

Für den Typ `char` ist der mögliche Gebrauch und damit auch die Vorzeichenregel zwiespältig:

- Wird `char` dazu verwendet einen **numerischen Wert** zu speichern und die Variable nicht explizit als vorzeichenbehaftet oder vorzeichenlos vereinbart, dann ist es implementierungsabhängig, ob `char` vorzeichenbehaftet ist oder nicht.
- Wenn ein Zeichen gespeichert wird, so garantiert der Standard, dass der gespeicherte Wert der nicht negativen Codierung im **Zeichensatz** entspricht.

```
char c = 'M';  // =  1001101 (ASCII Zeichensatz)
char c = 77;   // =  1001101
char s[] = "Eine kurze Zeichenkette";
```

Achtung: Anders als bei einigen anderen Programmiersprachen unterscheidet C/C++ zwischen den verschiedenen Anführungsstrichen.

ASCII Zeichensatz [ASCII]

[ASCII] ASCII-Tabelle

Sonderfall `bool`

Auf die Variablen von Datentyp `bool` können Werte `true` (1) und `false` (0) gespeichert werden. Eine implizite Umwandlung der ganzen Zahlen zu den Werten 0 und 1 ist ebenfalls möglich.

BoolExample.cpp

```
#include <iostream>

int main() {
    bool a = true;
    bool b = false;
    bool c = 45;

    std::cout<<"a = "<<a<<" b = "<<b<<" c = "<<c<<"\n";
    return 0;
}
```

```
Failed to execute 'send' on 'WebSocket': Still in CONNECTING state.
```

Sinnvoll sind boolsche Variablen insbesondere im Kontext von logischen Ausdrücken. Diese werden zum späteren Zeitpunkt eingeführt.

Architekturspezifische Ausprägung (Integer Datentypen)

Der Operator `sizeof` gibt Auskunft über die Größe eines Datentyps oder einer Variablen in Byte.

sizeof.cpp

```
#include <iostream>

int main(void)
{
  int x;
  std::cout<<"x umfasst " <<sizeof(x)<<" Byte.";
  return 0;
}
```

```
x umfasst 4 Byte.
```

sizeof_example.c

```c
#include <iostream>
#include <limits.h>   /* INT_MIN und INT_MAX */

int main(void) {
    std::cout<<"int size: "<< sizeof(int)<<" Byte\n";
    std::cout<<"Wertebereich von "<< INT_MIN<<" bis "<< INT_MAX<< "\n";
    std::cout<<"char size : "<< sizeof(char) <<" Byte\n";
    std::cout<<"Wertebereich von "<< CHAR_MIN<<" bis "<<CHAR_MAX<<"\n";
    return 0;
}
```

```
int size: 4 Byte
Wertebereich von -2147483648 bis 2147483647
char size : 1 Byte
Wertebereich von -128 bis 127
```

Die implementierungspezifische Werte, wie die Grenzen des Wertebereichs der ganzzahlinen Datentypen sind in `limits.h` definiert, z.B.

Makro	Wert
CHAR_MIN	-128
CHAR_MAX	+127
SHRT_MIN	-32768
SHRT_MAX	+32767
INT_MIN	-2147483648
INT_MAX	+2147483647
LONG_MIN	-9223372036854775808
LONG_MAX	+9223372036854775807

Was passiert bei der Überschreitung des Wertebereiches

> Der Arithmetische Überlauf (arithmetic overflow) tritt auf, wenn das Ergebnis einer Berechnung für den gültigen Zahlenbereich zu groß ist, um noch richtig interpretiert werden zu können.

Quelle: Arithmetischer Überlauf (Autor: WissensDürster)

Overflow.cpp

```cpp
#include <iostream>
#include <limits.h>   /* SHRT_MIN und SHRT_MAX */

int main(){
  short a = 30000;

  std::cout<<"Berechnung von 30000+3000 mit:\n\n";

  signed short c;     // -32768 bis 32767
  std::cout<<"(signed) short c - Wertebereich von "<<SHRT_MIN<<" bis "
    <<SHRT_MAX<<"\n";
  c = 3000 + a;       // ÜBERLAUF!
  std::cout<<"c="<<c<<"\n";

  unsigned short d;   //      0 bis 65535
  std::cout<<"unsigned short d - Wertebereich von "<<0<<" bis "<<USHRT_MAX
    <<"\n";

  d = 3000 + a;
  std::cout<<"d="<<d<<"\n";
}
```

```
Berechnung von 30000+3000 mit:

(signed) short c - Wertebereich von -32768 bis 32767
c=-32536
unsigned short d - Wertebereich von 0 bis 65535
d=33000
```

Ganzzahlüberläufe in der fehlerhaften Bestimmung der Größe eines Puffers oder in der Adressierung eines Feldes können es einem Angreifer ermöglichen den Stack zu überschreiben.

Fließkommazahlen

Fließkommazahlen sind Zahlen mit Nachkommastellen (reelle Zahlen). Im Gegensatz zu Ganzzahlen gibt es bei den Fließkommazahlen keinen Unterschied zwischen vorzeichenbehafteten und vorzeichenlosen Zahlen. Alle Fließkommazahlen sind in C/C++ immer vorzeichenbehaftet.

In C/C++ gibt es zur Darstellung reeller Zahlen folgende Typen:

Schlüsselwort	Mindestgröße
`float`	4 Byte
`double`	8 Byte
`long double`	je nach Implementierung

+

```
float <= double <=  long double
```

Gleitpunktzahlen werden halb logarithmisch dargestellt. Die Darstellung basiert auf die Zerlegung in drei Teile: ein Vorzeichen, eine Mantisse und einen Exponenten zur Basis 2.

Zur Darstellung von Fließkommazahlen sagt der C/C++-Standard nichts aus. Zur konkreten Realisierung ist die Headerdatei `float.h` auszuwerten.

	`float`	`double`
kleinste positive Zahl	1.1754943508e-38	2.2250738585072014E-308
Wertebereich	±3.4028234664e+38	±1.7976931348623157E+308

Achtung: Fließkommazahlen bringen einen weiteren Faktor mit - die Unsicherheit

float_precision.cpp

```cpp
#include<iostream>
#include<float.h>

int main(void) {
  std::cout<<"float Genauigkeit  :"<<FLT_DIG<<" \n";
  std::cout<<"double Genauigkeit :"<<DBL_DIG<<" \n";
  float x = 0.1;
  if (x == 0.1) {   // <- das ist ein double "0.1"
  //if (x == 0.1f) {   // <- das ist ein float "0.1"
    std::cout<<"Gleich\n";
  }else{
    std::cout<<"Ungleich\n";
  }
  return 0;
}
```

```
float Genauigkeit  :6
double Genauigkeit :15
Ungleich
```

Potenzen von 2 (zum Beispiel $2^{-3} = 0.125$) können im Unterschied zu `0.1` präzise im Speicher abgebildet werden. Können Sie erklären?

Datentyp `void`

`void` wird als „unvollständiger Typ“ bezeichnet, umfasst eine leere Wertemenge und wird verwendet überall dort, wo kein Wert vorhanden oder benötigt wird.

Anwendungsbeispiele:

- Rückgabewert einer Funktion
- Parameter einer Funktion
- anonymer Zeigertyp `void*`

```cpp
int main(void) {
  //Anweisungen
  return 0;
}
```

```cpp
void funktion(void) {
  //Anweisungen
}
```

Wertspezifikation

Zahlenliterale können in C/C++ mehr als Ziffern umfassen!

Gruppe	zulässige Zeichen
decimal-digits	`0` `1` `2` `3` `4` `5` `6` `7` `8` `9`
octal-prefix	`0`
octal-digits	`0` `1` `2` `3` `4` `5` `6` `7`
hexadecimal-prefix	`0x` `0X`
hexadecimal-digits	`0` `1` `2` `3` `4` `5` `6` `7` `8` `9`
	`a` `b` `c` `d` `e` `f`
	`A` `B` `C` `D` `E` `F`
unsigned-suffix	`u` `U`
long-suffix	`l` `L`
long-long-suffix	`ll` `LL`
fractional-constant	`.`
exponent-part	`e` `E`
binary-exponent-part	`p` `P`
sign	`+` `-`
floating-suffix	`f` `l` `F` `L`

Zahlentyp	Dezimal	Oktal	Hexadezimal
Eingabe	x	x	x
Ausgabe	x	x	x
Beispiel	`12`	`011`	`0x12`
	`0.123`		`0X1a`
	`123e-2`		`0xC.68p+2`
	`1.23F`		

Erkennen Sie jetzt die Bedeutung der Compilerfehlermeldung `error: invalid suffix "abc" on integer constant` aus dem ersten Beispiel der Vorlesung?

`Variable = (Vorzeichen)(Zahlensystem)[Wert](Typ);`

Literal	Bedeutung
12	Ganzzahl vom Typ `int`
-234L	Ganzzahl vom Typ `signed long`
100000000000	Ganzzahl vom Typ `long`
011	Ganzzahl also oktale Zahl (Wert 9_d)
0x12	Ganzzahl (18_d)
1.23F	Fließkommazahl vom Typ `float`
0.132	Fließkommazahl vom Typ `double`
123e-2	Fließkommazahl vom Typ `double`
0xC.68p+2	hexadizimale Fließkommazahl vom Typ `double`

NumberFormats.cpp

```cpp
#include<iostream>

int main(void)
{
  int x=020;
  int y=0x20;
  std::cout<<"x = "<<x<<"\n";
  std::cout<<"y = "<<y<<"\n";
  std::cout<<"Rechnen mit Oct und Hex x + y = "<< x + y;
  return 0;
}
```

Adressen

Merke: Einige Anweisungen in C/C++ verwenden Adressen von Variablen.

Jeder Variable in C++ wird eine bestimmten Position im Hauptspeicher zugeordnet. Diese Position nennt man Speicheradresse. Solange eine Variable gültig ist, bleibt sie an dieser Stelle im Speicher. Um einen Zugriff auf die Adresse einer Variablen zu haben, kann man den Operator `&` nutzen.

Pointer.cpp

```cpp
#include <iostream>

int main(void)
{
  int x=020;
  std::cout<<&x<<"\n";
  return 0;
}
```

Sichtbarkeit und Lebensdauer von Variablen

Lokale Variablen

Variablen *leben* innerhalb einer Funktion, einer Schleife oder einfach nur innerhalb eines durch geschwungene Klammern begrenzten Blocks von Anweisungen von der Stelle ihrer Definition bis zum Ende des Blocks. Beachten Sie, dass die Variable vor der ersten Benutzung vereinbart werden muss.

Wird eine Variable/Konstante z. B. im Kopf einer Schleife vereinbart, gehört sie zu dem Block, in dem auch der Code der Schleife steht. Folgender Codeausschnitt soll das verdeutlichen:

visibility.cpp

```cpp
#include<iostream>

int main(void)
{
  int v = 1;
  int w = 5;
  {
    int v;
    v = 2;
    std::cout<<v<<"\n";
    std::cout<<w<<"\n";
  }
  std::cout<<v<<"\n";
  return 0;
}
```

Globale Variablen

Muss eine Variable immer innerhalb von `main` definiert werden? Nein, allerdings sollten globale Variablen vermieden werden.

visibility.cpp

```cpp
#include<iostream>

int v = 1; /*globale Variable*/

int main(void)
{
  std::cout<<v<<"\n";
  return 0;
}
```

Sichtbarkeit und Lebensdauer spielen beim Definieren neuer Funktionen eine wesentliche Rolle und werden in einer weiteren Vorlesung in diesem Zusammenhang nochmals behandelt.

Definition vs. Deklaration vs. Initialisierung

... oder andere Frage, wie kommen Name, Datentyp, Adresse usw. zusammen?

> Deklaration ist nur die Vergabe eines Namens und eines Typs für die Variable. Definition ist die Reservierung des Speicherplatzes. Initialisierung ist die Zuweisung eines ersten Wertes.

> **Merke:** Jede Definition ist gleichzeitig eine Deklaration aber nicht umgekehrt!

DeclarationVSDefinition.cpp

```cpp
extern int a;       // Deklaration
int i;              // Definition + Deklaration
int a,b,c;
int i = 5;          // Definition + Deklaration + Initialisierung
```

Das Schlüsselwort `extern` in obigem Beispiel besagt, dass die Definition der Variablen `a` irgendwo in einem anderen Modul des Programms liegt. So deklariert man Variablen, die später beim Binden (Linken) aufgelöst werden. Da in diesem Fall kein Speicherplatz reserviert wurde, handelt es sich um keine Definition.

Typische Fehler

Fehlende Initialisierung

MissingInitialisation.cpp

```cpp
#include<iostream>

int main(void) {
  int x = 5;
  std::cout<<"x="<<x<<"\n";
  int y;     // <- Fehlende Initialisierung
  std::cout<<"y="<<y<<"\n";
  return 0;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:7:23: warning: ‘y’ is used uninitialized [-Wuninitialized]
    7 |   std::cout<<"y="<<y<<"\n";
      |                       ^~~~
main.cpp:6:7: note: ‘y’ was declared here
    6 |   int y;     // <- Fehlende Initialisierung
      |       ^
```

Redeklaration

Redeclaration.cpp

```cpp
#include<iostream>

int main(void) {
  int x;
  int x;
  return 0;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:5:7: error: redeclaration of ‘int x’
    5 |   int x;
      |       ^
main.cpp:4:7: note: ‘int x’ previously declared here
    4 |   int x;
      |       ^
main.cpp:4:7: warning: unused variable ‘x’ [-Wunused-variable]
```

Falsche Zahlenliterale

wrong_float.cpp

```cpp
#include<iostream>

int main(void) {
  float a=1,5;   /* FALSCH  */
  float b=1.5;   /* RICHTIG */
  return 0;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:4:13: error: expected unqualified-id before numeric constant
    4 |   float a=1,5;   /* FALSCH  */
      |             ^
main.cpp:4:9: warning: unused variable ‘a’ [-Wunused-variable]
    4 |   float a=1,5;   /* FALSCH  */
      |         ^
main.cpp:5:9: warning: unused variable ‘b’ [-Wunused-variable]
    5 |   float b=1.5;   /* RICHTIG */
      |         ^
```

Was passiert wenn der Wert zu groß ist?

TooLarge.cpp

```cpp
#include<iostream>

int main(void) {
  short a;
  a = 0xFFFF + 2;
  std::cout<<"Schaun wir mal ... "<<a<<"\n";
  return 0;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:5:14: warning: overflow in conversion from ‘int’ to ‘short int’ changes value from ‘65537’ to ‘1’ [-Woverflow]
    5 |   a = 0xFFFF + 2;
      |       ~~~~~~~^~~
```

Ein- und Ausgabe

Ausgabefunktionen wurden bisher genutzt, um den Status unserer Programme zu dokumentieren. Nun soll dieser Mechanismus systematisiert und erweitert werden.

Quelle: EVA-Prinzip (Autor: Deadlyhappen)

Für Ein- und Ausgabe stellt C++ das Konzept der Streams bereit, dass nicht nur für elementare Datentypen gilt, sondern auch auf die neu definierten Datentypen (Klassen) erweitert werden kann. Unter Stream wird eine Folge von Bytes verstanden.

Als Standard werden verwendet:

- `std::cin` für die Standardeingabe (Tastatur),
- `std::cout` für die Standardausgabe (Console) und
- `std::cerr` für die Standardfehlerausgabe (Console)

Achtung: Das `std::` ist ein zusätzlicher Indikator für eine bestimmte Implementierung, ein sogenannter Namespace. Um sicherzustellen, dass eine spezifische Funktion, Datentyp etc. genutzt wird stellt man diese Bezeichnung dem verwendenten Element zuvor. Mit `using namespace std;` kann man die permanente Nennung umgehen.

Stream-Objekte werden durch `#include <iostream>` bekannt gegeben. Definiert werden sie als Komponente der Standard Template Library (STL) im Namensraum `std`.

Mit Namensräumen können Deklarationen und Definitionen unter einem Namen zusammengefasst und gegen andere Namen abgegrenzt werden.

iostream.cpp

```cpp
#include <iostream>

int main(void) {
  char hanna[]="Hanna";
  char anna[]="Anna";
  std::cout << "C++ stream: " << "Hallo "<< hanna << ", " << anna <<std
    ::endl;
  return 0;
}
```

Ausgabe

Der Ausgabeoperator `<<` formt automatisch die Werte der Variablen in die Textdarstellung der benötigten Weite um. Der Rückgabewert des Operators ist selbst ein Stream-Objekt (Referenz), so dass ein weiterer Ausgabeoperator darauf angewendet werden kann. Damit ist die Hintereinanderschaltung von Ausgabeoperatoren möglich.

```cpp
std::cout<<55<<"55"<<55.5<<true;
```

Welche Formatierungmöglichkeiten bietet der Ausgabeoperator noch?

Mit Hilfe von in `<iomanip>` definierten Manipulatoren können besondere Ausgabeformatierungen erreicht werden.

Manipulator	Bedeutung
`setbase(int B)`	Basis 8, 10 oder 16 definieren
`setfill(char c)`	Füllzeichen festlegen
`setprecision(int n)`	Flieskommaprezession
`setw(int w)`	Breite setzen

manipulatoren1.c

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

int main(){
  std::cout<<std::setbase(16)<< std::fixed<<55<<std::endl;
  std::cout<<std::setbase(10)<< std::fixed<<55<<std::endl;
  return 0;
}
```

Achtung: Die Manipulatoren wirken auf alle daruf folgenden Ausgaben.

Feldbreite

Die Feldbreite definiert die Anzahl der nutzbaren Zeichen, sofern diese nicht einen größeren Platz beanspruchen.

Der Manipulator `right` sorgt im Beispiel für eine rechtsbündige Ausrichtung der Ausgabe, wegen `setw(5)` ist die Ausgabe fünf Zeichen breit, wegen `setfill('0')` werden nicht benutzte Stellen mit dem Zeichen 0 aufgefüllt, `endl` bewirkt die Ausgabe eines Zeilenumbruchs.

manipulatoren2.c

```
#include <iostream>
#include <iomanip>
int main(){

  std::cout<<std::right<< std::setw(5)<<55<<std::endl;
  std::cout<<std::right<< std::setfill('0')<<std::setw(5)<<55<<std::endl;
  std::cout<<std::left<< std::fixed<<std::setw(5)<<55<<std::endl;
  std::cout<<std::setw(5)<<"Zu klein gedacht: "<<234534535<<std::endl;
  return 0;
}
```

Genauigkeit

genauigkeit.cpp

```
#include <iostream>
#include <iomanip>
#include <math.h>

int main() {
  for (int i = 12; i > 1; i -=3) {
    std::cout << std::setprecision(i) << std::fixed << M_PI << std::endl;
  }
}
```

Escape-Sequenzen

Sequenz	Bedeutung
`\n`	newline
`\b`	backspace
`\r`	carriage return
`\t`	horizontal tab
`\\`	backslash
`\'`	single quotation mark
`\"`	double quotation mark

esc_sequences.c

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

int main(){
  cout << "123456789\r";
  cout << "ABCD\n\n";
  cout << "Vorname \t Name \t\t Alter \n";
  cout << "Andreas \t Mustermann\t 42 \n\n";
  cout << "Manchmal braucht man auch ein \"\\\"\n";
  return 0;
}
```

Eingabe

Für die Eingabe stellt iostream den Eingabeoperator `>>` zur Verfügung. Der Rückgabewert des Operators ist ebenfalls eine Referenz auf ein Stream-Objekt (Referenz), so dass auch hier eine Hintereinanderschaltung von Operatoren möglich ist.

istream.cpp

```c
#include <iostream>

int main()
{
  char b;
  float a;
  int i;
  std::cout<<"Bitte Werte eingeben [char float int] : ";
  std::cin>>b>>a>>i;
  std::cout<<"char - " <<b<< " float - "<<a<<" int - "<<i;
  return 0;
}
```

Beispiel der Woche

Implementieren Sie einen programmierbaren Taschenrechner für quadaratische Funktionen.

QuadraticEquation.cpp

```cpp
#include<iostream>

int main() {
  // Variante 1 - ganz schlecht
  std::cout <<"f("<<x<<") = "<<3*5*5 + 4*5 + 8<<" \n";

  // Variante 2 - besser
  int x = 9;
  std::cout <<"f("<<x<<") = "<<3*x*x + 4*x + 8<<" \n";
  return 0;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:5:21: error: ‘x’ was not declared in this scope
    5 |   std::cout <<"f("<<x<<") = "<<3*5*5 + 4*5 + 8<<" \n";
      |                     ^
```