

4. Wahrheitswerte

Boolesche Funktionen; der Typ `bool`; logische und relationale Operatoren; Kurzschlussauswertung

Wo wollen wir hin?

```
int a;  
std::cin >> a;  
if (a % 2 == 0)  
    std::cout << "even";  
else  
    std::cout << "odd";
```

Verhalten hängt ab vom Wert eines **Booleschen Ausdrucks**

Boolesche Werte in der Mathematik

Boolesche Ausdrücke können zwei mögliche Werte annehmen:

0 oder *1*

- *0* entspricht „*falsch*“
- *1* entspricht „*wahr*“

Der Typ `bool` in C++

- Repräsentiert *Wahrheitswerte*
- Literale `false` und `true`
- Wertebereich `{false, true}`

```
bool b = true; // Variable mit Wert true (wahr)
```

- a < b (kleiner als)
- a >= b (grösser gleich)
- a == b (gleich)
- a != b (ungleich)

Zahlentyp × Zahlentyp → bool

R-Wert × R-Wert → R-Wert

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Kleiner	<	2	11	links
Grösser	>	2	11	links
Kleiner gleich	<=	2	11	links
Grösser gleich	>=	2	11	links
Gleich	==	2	10	links
Ungleich	!=	2	10	links

Zahlentyp × Zahlentyp → bool

R-Wert × R-Wert → R-Wert

164

165

Boolesche Funktionen in der Mathematik

- Boolesche Funktion

$$f : \{0, 1\}^2 \rightarrow \{0, 1\}$$

- 0 entspricht „falsch“.
- 1 entspricht „wahr“.

AND(x, y)

$$x \wedge y$$

- “Logisches Und”

$$f : \{0, 1\}^2 \rightarrow \{0, 1\}$$

- 0 entspricht „falsch“.
- entspricht „wahr“.

x	y	AND(x, y)
0	0	0
0	1	0
1	0	0
1	1	1

Logischer Operator &&

`a && b` (logisches Und)

`bool × bool → bool`

R-Wert × R-Wert → R-Wert

```
int n = -1;
int p = 3;
bool b = (n < 0) && (0 < p); // b = true (wahr)
```

OR(x, y)

$x \vee y$

- "Logisches Oder"

$$f : \{0, 1\}^2 \rightarrow \{0, 1\}$$

- 0 entspricht „falsch“.
- entspricht „wahr“.

x	y	OR(x, y)
0	0	0
0	1	1
1	0	1
1	1	1

Logischer Operator ||

`a || b` (logisches Oder)

`bool × bool → bool`

R-Wert × R-Wert → R-Wert

```
int n = 1;
int p = 0;
bool b = (n < 0) || (0 < p); // b = false (falsch)
```

NOT(x)

$\neg x$

- "Logisches Nicht"

$$f : \{0, 1\} \rightarrow \{0, 1\}$$

- 0 entspricht „falsch“.
- entspricht „wahr“.

x	NOT(x)
0	1
1	0

!b (logisches Nicht)

bool → bool
R-Wert → R-Wert

```
int n = 1;
bool b = !(n < 0); // b = true (wahr)
```

$$\begin{array}{c} !b \ \&\& \ a \\ \Downarrow \\ (!b) \ \&\& \ a \\ \\ a \ \&\& \ b \ || \ c \ \&\& \ d \\ \Downarrow \\ (a \ \&\& \ b) \ || \ (c \ \&\& \ d) \\ \\ a \ || \ b \ \&\& \ c \ || \ d \\ \Downarrow \\ a \ || \ (b \ \&\& \ c) \ || \ d \end{array}$$

172

173

Logische Operatoren: Tabelle

	Symbol	Stelligkeit	Präedenz	Assoziativität
Logisches Und (AND)	&&	2	6	links
Logisches Oder (OR)		2	5	links
Logisches Nicht (NOT)	!	1	16	rechts

Präedenzen

Der *unäre logische* Operator !

bindet stärker als

binäre arithmetische Operatoren. Diese

binden stärker als

relationale Operatoren,

und diese binden stärker als

binäre logische Operatoren.

```
7 + x < y && y != 3 * z || ! b
7 + x < y && y != 3 * z || (!b)
```

174

175

- AND, OR und NOT sind die in C++ verfügbaren Booleschen Funktionen.
- Alle anderen *binären* Booleschen Funktionen sind daraus erzeugbar.

x	y	XOR(x, y)
0	0	0
0	1	1
1	0	1
1	1	0

$$\text{XOR}(x, y) = \text{AND}(\text{OR}(x, y), \text{NOT}(\text{AND}(x, y))).$$

$$x \oplus y = (x \vee y) \wedge \neg(x \wedge y).$$

$$(x \ || \ y) \ \&\& \ ! (x \ \&\& \ y)$$

176

177

Vollständigkeit Beweis

- Identifiziere binäre Boolesche Funktionen mit ihrem charakteristischen Vektor.

x	y	XOR(x, y)
0	0	0
0	1	1
1	0	1
1	1	0

Charakteristischer Vektor: **0110**

$$\text{XOR} = f_{0110}$$

Vollständigkeit Beweis

- Schritt 1: erzeuge die *elementaren* Funktionen $f_{0001}, f_{0010}, f_{0100}, f_{1000}$

$$f_{0001} = \text{AND}(x, y)$$

$$f_{0010} = \text{AND}(x, \text{NOT}(y))$$

$$f_{0100} = \text{AND}(y, \text{NOT}(x))$$

$$f_{1000} = \text{NOT}(\text{OR}(x, y))$$

178

179

- Schritt 2: erzeuge alle Funktionen durch "Veroderung" elementarer Funktionen

$$f_{1101} = \text{OR}(f_{1000}, \text{OR}(f_{0100}, f_{0001}))$$

- Schritt 3: erzeuge f_{0000}

$$f_{0000} = 0.$$

- bool kann überall dort verwendet werden, wo int gefordert ist – und umgekehrt.
- Viele existierende Programme verwenden statt bool den Typ int.
Das ist schlechter Stil, der noch auf die Sprache C zurückgeht.

bool	→	int
true	→	1
false	→	0
int	→	bool
≠0	→	true
0	→	false

`bool b = 3; // b=true`

DeMorgansche Regeln

- $!(a \ \&\& \ b) == (!a \ || \ !b)$
- $!(a \ || \ b) == (!a \ \&\& \ !b)$

! (reich *und* schön) == (arm *oder* hässlich)

Anwendung: Entweder ... Oder (XOR)

$(x \ || \ y) \ \&\& \ !(x \ \&\& \ y)$ x oder y, und nicht beide

$(x \ || \ y) \ \&\& \ (!x \ || \ !y)$ x oder y, und eines nicht

$!(!x \ \&\& \ !y) \ \&\& \ !(x \ \&\& \ y)$ nicht keines, und nicht beide

$!(!x \ \&\& \ !y \ || \ x \ \&\& \ y)$ nicht: keines oder beide

- Logische Operatoren `&&` und `||` werten den *linken Operanden* *zuerst* aus.
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet.

```
x != 0 && z / x > y
```

⇒ Keine Division durch 0

- Fehler, die der Compiler findet: syntaktische und manche semantische Fehler
- Fehler, die der Compiler nicht findet: Laufzeitfehler (immer semantisch)

184

1. Genaue Kenntnis des gewünschten Programmverhaltens

```
>> It's not a bug, it's a feature !!<<
```
2. Überprüfe an vielen kritischen Stellen, ob das Programm auf dem richtigen Weg ist!
3. Hinterfrage auch das (scheinbar) Offensichtliche, es könnte sich ein simpler Tippfehler eingeschlichen haben!

186

```
assert(expr)
```

- hält das Programm an, falls der boolesche Ausdruck `expr` nicht wahr ist
- benötigt `#include <cassert>`
- kann abgeschaltet werden

185

187

DeMorgansche Regeln

Hinterfrage das Offensichtliche! Hinterfrage das **scheinbar** Offensichtliche!

```
// Prog: assertion.cpp
// use assertions to check De Morgan's laws

#include<cassert>

int main()
{
    bool x; // whatever x and y actually are,
    bool y; // De Morgan's laws will hold:
    assert ( !(x && y) == (x || !y) );
    assert ( !(x || y) == (!x && !y) );
    return 0;
}
```

Assertions abschalten

```
// Prog: assertion2.cpp
// use assertions to check De Morgan's laws. To tell the
// compiler to ignore them, #define NDEBUG ("no debugging")
// at the beginning of the program, before the #includes

#define NDEBUG
#include<cassert>

int main()
{
    bool x; // whatever x and y actually are,
    bool y; // De Morgan's laws will hold:
    assert ( !(x && y) == (x || !y) ); // ignored by NDEBUG
    assert ( !(x || y) == (!x && !y) ); // ignored by NDEBUG
    return 0;
}
```

Div-Mod Identität

$$a/b * b + a\%b == a$$

Überprüfe, ob das Programm auf dem richtigen Weg ist...

```
std::cout << "Dividend a =? ";
int a;
std::cin >> a;
```

Eingabe der Argumente für die Berechnung

```
std::cout << "Divisor b =? ";
int b;
std::cin >> b;
```

```
// check input
assert ( b != 0 );
```

Vorbedingung für die weitere Berechnung

Div-Mod Identität

$$a/b * b + a\%b == a$$

... und hinterfrage das Offensichtliche!

```
// check input
assert ( b != 0 );
```

Vorbedingung für die weitere Berechnung

```
// compute result
int div = a / b;
int mod = a % b;
```

```
// check result
assert ( div * b + mod == a );
```

Div-Mod Identität

...

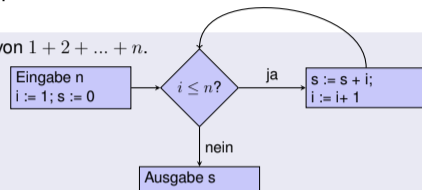
5. Kontrollanweisungen I

Auswahlweisungen, Iterationsanweisungen, Terminierung, Blöcke

Kontrollfluss

- bisher *linear* (von oben nach unten)
- Für interessante Programme braucht man „Verzweigungen“ und „Sprünge“.

Berechnung von $1 + 2 + \dots + n$.



Auswahlweisungen

realisieren Verzweigungen

- if Anweisung
- if-else Anweisung

if-Anweisung

```
if ( condition )  
    statement
```

```
int a;  
std::cin >> a;  
if ( a % 2 == 0 )  
    std::cout << "even";
```

Ist *condition* wahr, dann wird *statement* ausgeführt.

- *statement*: beliebige Anweisung (*Rumpf* der if-Anweisung)
- *condition*: konvertierbar nach bool

if-else-Anweisung

```
if ( condition )
    statement1
else
    statement2
```

```
int a;
std::cin >> a;
if ( a % 2 == 0 )
    std::cout << "even";
else
    std::cout << "odd";
```

Ist *condition* wahr, so wird *statement1* ausgeführt, andernfalls wird *statement2* ausgeführt.

- *condition*: konvertierbar nach bool.
- *statement1*: Rumpf des if-Zweiges
- *statement2*: Rumpf des else-Zweiges

Layout!

```
int a;
std::cin >> a;
if ( a % 2 == 0 )
    std::cout << "even"; ← Einrückung
else
    std::cout << "odd"; ← Einrückung
```

Iterationsanweisungen

realisieren „Schleifen“:

- for-Anweisung
- while-Anweisung
- do-Anweisung

Berechne $1 + 2 + \dots + n$

```
// Program: sum_n.cpp
// Compute the sum of the first n natural numbers.

#include <iostream>

int main()
{
    // input
    std::cout << "Compute the sum 1+...+n for n =? ";
    unsigned int n;
    std::cin >> n;

    // computation of sum_{i=1}^n i
    unsigned int s = 0;
    for (unsigned int i = 1; i <= n; ++i) s += i;

    // output
    std::cout << "1+...+" << n << " = " << s << ".\n";
    return 0;
}
```

for-Anweisung am Beispiel

```
for (unsigned int i=1; i <= n ; ++i)
    s += i;
```

Annahmen: $n == 2, s == 0$

i		s
i==1	wahr	s == 1
i==2	wahr	s == 3
i==3	falsch	

s == 3

for-Anweisung: Syntax

```
for ( init statement condition ; expression )
    statement
```

- *init-statement*: Ausdrucksanweisung, Deklarationsanweisung, Nullanweisung
- *condition*: konvertierbar nach bool
- *expression*: beliebiger Ausdruck
- *statement*: beliebige Anweisung (*Rumpf* der for-Anweisung)

200

for-Anweisung: Semantik

```
for ( init statement condition ; expression )
    statement
```

- *init-statement* wird ausgeführt
- *condition* wird ausgewertet
 - true: Iteration beginnt
statement wird ausgeführt
expression wird ausgeführt
 - falsch: for-Anweisung wird beendet.

Der kleine Gauß (1777 - 1855)

- Mathe-Lehrer wollte seine Schüler mit folgender Aufgabe beschäftigen:

Berechne die Summe der Zahlen von 1 bis 100 !

- Gauß war nach einer Minute fertig.

201

202

Die Lösung von Gauß

- Die gesuchte Zahl ist

$$1 + 2 + 3 + \dots + 98 + 99 + 100.$$

- Das ist die Hälfte von

$$\begin{array}{r} 1 + 2 + \dots + 99 + 100 \\ + 100 + 99 + \dots + 2 + 1 \\ \hline = 101 + 101 + \dots + 101 + 101 \end{array}$$

- Antwort: $100 \cdot 101/2 = 5050$

for-Anweisung: Terminierung

```
for (unsigned int i = 1; i <= n; ++i)
    s += i;
```

Hier und meistens:

- expression* ändert einen Wert, der in *condition* vorkommt.
- Nach endlich vielen Iterationen wird *condition* falsch: *Terminierung*.

Endlosschleifen

- Endlosschleifen sind leicht zu produzieren:

```
for ( ; ; ) ;
```

- Die *leere condition* ist wahr.
 - Die *leere expression* hat keinen Effekt.
 - Die *Nullanweisung* hat keinen Effekt.
- ... aber nicht automatisch zu erkennen.

```
for ( e; v; e) r;
```

Halteproblem

Unentscheidbarkeit des Halteproblems

Es gibt kein C++ Programm, das für jedes C++- Programm P und jede Eingabe I korrekt feststellen kann, ob das Programm P bei Eingabe von I terminiert.

Das heisst, die Korrektheit von Programmen kann *nicht* automatisch überprüft werden.⁵

⁵Alan Turing, 1936. Theoretische Fragestellungen dieser Art waren für Alan Turing die Hauptmotivation für die Konstruktion seiner Rechenmaschine.

Def.: Eine natürliche Zahl $n \geq 2$ ist eine Primzahl, wenn kein $d \in \{2, \dots, n-1\}$ ein Teiler von n ist.

Eine Schleife, die das testet:

```
unsigned int d;  
for (d=2; n%d != 0; ++d);
```

- Beobachtung 1:
Nach der `for`-Anweisung gilt $d \leq n$.
- Beobachtung 2:
 n ist Primzahl genau wenn am Ende $d = n$.

- Blöcke gruppieren mehrere Anweisungen zu einer neuen Anweisung

```
{statement1 statement2 ... statementN}
```

- Beispiel: Rumpf der main Funktion

```
int main() {  
    ...  
}
```

- Beispiel: Schleifenrumpf

```
for (unsigned int i = 1; i <= n; ++i) {  
    s += i;  
    std::cout << "partial sum is " << s << "\n";  
}
```