3. Wahrheitswerte

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

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 Funktionen

- Boolesche Funktion

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

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

- 0 entspricht „falsch”.
- 1 entspricht „wahr”.
- $\mathrm{AND}(x,y) = x \wedge y$
- $\mathrm{OR}(x,y) = x \vee y$
- $\mathrm{NOT}(x) = \neg x$

x	y	AND	OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

x	$\mathrm{NOT}(x)$
0	1
1	0

Vollständigkeit

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

x	y	$\mathrm{XOR}(x, y)$
0	0	0
0	1	1
1	0	1
1	1	0

Vollständigkeit: $\mathrm{XOR}(x, y)$ $x \oplus y$

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

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

Vollständigkeit Beweis

- Identifiziere binäre Boolesche Funktionen mit ihrem charakteristischen Vektor.

x	y	$\mathrm{XOR}(x, y)$
0	0	0
0	1	1
1	0	1
1	1	0

Charakteristischer Vektor: 0110

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

Vollständigkeit Beweis

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

$$
\begin{aligned}
f_{0001} &= \mathrm{AND}(x, y) \\
f_{0010} &= \mathrm{AND}(x, \mathrm{NOT}(y)) \\
f_{0100} &= \mathrm{AND}(y, \mathrm{NOT}(x)) \\
f_{1000} &= \mathrm{NOT}(\mathrm{OR}(x, y))
\end{aligned}
$$

Vollständigkeit Beweis

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

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

- Schritt 3: erzeuge f_{0000}

$$f_{0000} = 0.$$

Der Typ `bool`

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

```
bool b = true; // Variable mit Wert true
```

`bool` vs `int`: Konversion

- **`bool`** kann (leider) überall dort verwendet werden, wo **`int`** gefordert ist – und umgekehrt.
- Viele existierende Programme verwenden statt **`bool`** oft nur die **`int`**-Werte 0 und 1.
  Das ist schlechter Stil, der noch auf die Sprache C zurückgeht.

`bool`	$\rightarrow$	**`int`**
true	$\rightarrow$	1
false	$\rightarrow$	0
`int`	$\rightarrow$	**`bool`**
$\neq$**`0`**	$\rightarrow$	true
`0`	$\rightarrow$	false

Logische Operatoren

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

`bool` ($\times$ **`bool`**) $\rightarrow$ **`bool`**

R-Wert ($\times$ R-Wert) $\rightarrow$ R-Wert

Relationale Operatoren

	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 $\times$ Zahlentyp $\rightarrow$ `bool`

R-Wert $\times$ R-Wert $\rightarrow$ R-Wert

DeMorgan'sche Regeln

- `!(a && b) == (!a || !b)`
- `!(a || b) == (!a && !b)`

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

Präzedenzen

Binäre Arithmetische Operatoren

binden stärker als

relationale Operatoren

und diese binden stärker als

binäre logische Operatoren.

```
7 + x < y && y != 3 * z
```

Kurzschlussauswertung

- 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
```

$\Rightarrow$ Keine Division durch 0

Assertions: Fehlerquellen

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

Assertions: Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens

   ≫ It's not a bug, it's a feature !!≪

2. Überprüfe an vielen kritischen Stellen, dass das Programm auf dem richtigen Weg ist

Assertions

```
assert(expr)
```

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

Assertions, Beispiel De Morgansches Gesetz

```
// 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, Beispiel De Morgansches Gesetz

```
// 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;
}
```

Assertions, Beispiel divmod Identität

```
// Prog: divmod.cpp
// demonstrates the law of integer division with
// remainder: a / b * b + a % b == a, for b != 0

#include<iostream>
#include<cassert>

int main ()
{
  // input a and b
  std::cout << "Dividend a =? ";
  int a;
  std::cin >> a;

  std::cout << "Divisor b =? ";
  int b;
  std::cin >> b;
  assert (b != 0);

  // check the law for a and b
  assert (a / b * b + a % b == a);

  // print the law for a and b
  std::cout << a << " / " << b << " * " << b
```

Konstanten

- sind Variablen mit unveränderbarem Wert

```
const int speed_of_light = 299792458;
```

- Verwendung: **`const`** vor der Definition
- Compiler kontrolliert Einhaltung des **`const`**-Versprechens

```
const int speed_of_light = 299792458;
...
speed_of_light = 300000000;
```

Compilerfehler!

- Hilfsmittel zur Vermeidung von Fehlern: Konstanten erlauben garantierte Einhaltung der Invariante *Wert ändert sich nicht*

Die `const`-Richtlinie

`const`-Richtlinie

Denke bei *jeder Variablen* darüber nach, ob sie im Verlauf des Programmes jemals ihren Wert ändern wird oder nicht! Im letzteren Falle verwende das Schlüsselwort **`const`**, um die Variable zu einer Konstanten zu machen!

Ein Programm, welches diese Richtlinie befolgt, heisst **`const`**-korrekt.

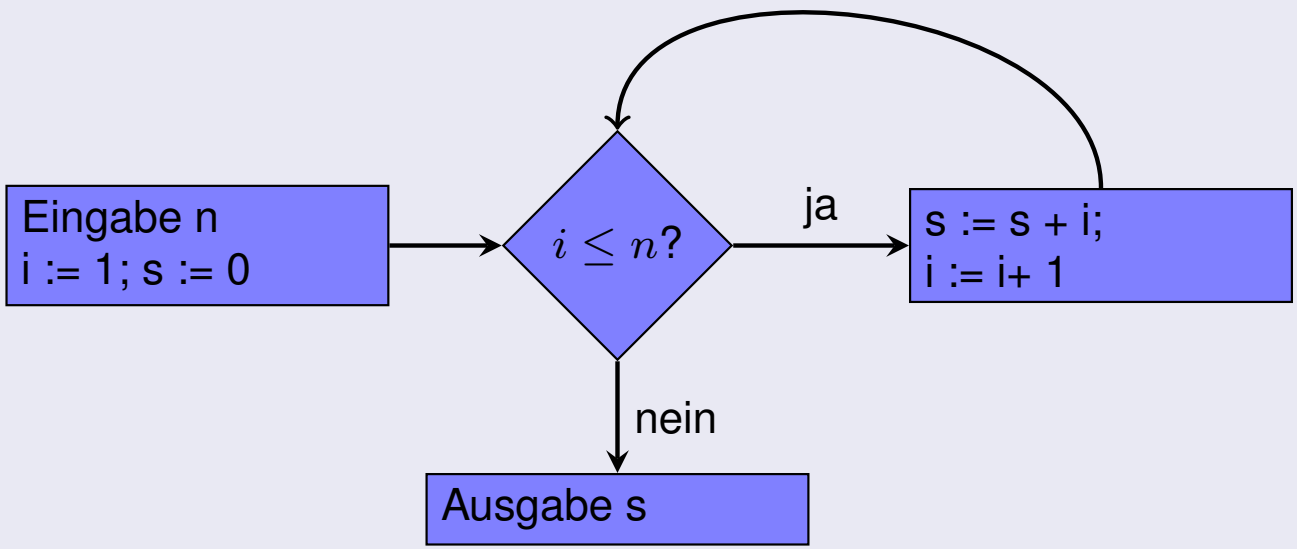
4. Kontrollanweisungen

Auswahlanweisungen, Iterationsanweisungen, Terminierung, Blöcke

Kontrollfluss

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

Berechung von $1 + 2 + ... + n$.



Auswahlanweisungen

realisieren Verzweigungen

- `if` Anweisung
- `if-else` Anweisung

`if`-Anweisung

`if` (*condition*)
statement

Wenn *condition* den Wert **`true`** hat, dann wird *statement* ausgeführt.

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

`if-else`-Anweisung

`if` (*condition*)
    statement1
else
    statement2

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

Iterationsanweisungen

realisieren „Schleifen”:

- **`for`**-Anweisung
- **`while`**-Anweisung
- **`do`**-Anweisung

Berechne $1 + 2 + \ldots + n$

```
// 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: Illustration

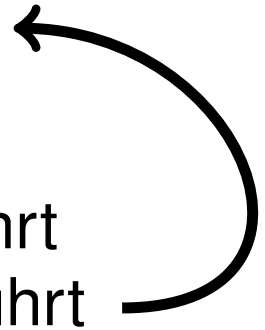
`for` (*init statement* *condition* **`;`** *expression*)
statement

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

`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
  - **`false`**: **`for`**-Anweisung wird beendet.



`for`-Anweisung formal

`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)

`for`-Anweisung: Beispiel

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

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

`i`		`s`
`i==1`	`true`	`s == 1`
`i==2`	`true`	`s == 3`
`i==3`	`false`	
		`s == 3`

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.

Die Lösung von Gauß

- Die gesuchte Zahl ist

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

- Das ist die Hälfte von

$$\begin{array}{rcrcccrcr} & 1 & + & 2 & + & \cdots & + & 99 & + & 100 \\ + & 100 & + & 99 & + & \cdots & + & 2 & + & 1 \\ \hline = & 101 & + & 101 & + & \cdots & + & 101 & + & 101 \end{array}$$

- 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 hat *condition* den Wert **`false`**: *Terminierung*.

Endlosschleifen

- Endlosschleifen sind leicht zu produzieren:

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

  - Die *leere condition* hat den Wert `true`.
  - 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 $\mathrm{C{+}{+}}$ Programm, das für jedes $\mathrm{C{+}{+}}$-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.[4]

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

Beispiel: Primzahltest

Def.: Eine natürliche Zahl $n \geq 2$ ist eine Primzahl, wenn kein $d \in \{2, \ldots, n-1\}$ 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 $d = n$.

Beispiel: Primzahltest

```
// Program: prime.cpp
// Test if a given natural number is prime.

#include <iostream>
#include <cassert>

int main ()
{
  // Input
  unsigned int n;
  std::cout << "Test if n>1 is prime for n =? ";
  std::cin >> n;
  assert (n > 1);

  // Computation: test possible divisors d
  unsigned int d;
  for (d = 2; n % d != 0; ++d);

  // Output
  if (d < n)
    // d is a divisor of n in {2,...,n-1}
    std::cout << n << " = " << d << " * " << n / d << ".\n";
  else {
    // no proper divisor found
    assert (d == n);
    std::cout << n << " is prime.\n";
  }
```

Blöcke

- 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";
}
```