3. 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";
```

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*

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**

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**

Der Typ `bool` in C++

- Repräsentiert **Wahrheitswerte**
- Literale **`false`** und **`true`**

Der Typ `bool` in C++

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

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

Relationale Operatoren

`a < b` (kleiner als)

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

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

Relationale Operatoren

`a < b` (kleiner als)

```
bool b = (1 < 3); // b =
```

Relationale Operatoren

`a < b` (kleiner als)

```
bool b = (1 < 3); // b = true (wahr)
```

Relationale Operatoren

`a >= b` (grösser gleich)

```
int a = 0;
bool b = (a >= 3); // b =
```

Relationale Operatoren

`a >= b` (grösser gleich)

```
int a = 0;
bool b = (a >= 3); // b = false (falsch)
```

Relationale Operatoren

`a == b` (gleich)

```
int a = 4;
bool b = (a % 3 == 1); // b =
```

Relationale Operatoren

`a == b` (gleich)

```
int a = 4;
bool b = (a % 3 == 1); // b = true (wahr)
```

Relationale Operatoren

`a != b` (ungleich)

```
int a = 1;
bool b = (a != 2*a-1); // b =
```

Relationale Operatoren

`a != b` (ungleich)

```
int a = 1;
bool b = (a != 2*a-1); // b = false (falsch)
```

Boolesche Funktionen in der Mathematik

- Boolesche Funktion

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

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

$\mathrm{AND}(x, y)$

$x \wedge y$

- „Logisches Und“

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

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

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

Logischer Operator &&

a && b (logisches Und)

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

Logischer Operator &&

a && b (logisches Und)

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

$\mathrm{OR}(x, y)$

$x \vee y$

- „Logisches Oder“

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

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

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

Logischer Operator ||

`a || b` (logisches Oder)

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

Logischer Operator ||

`a || b` (logisches Oder)

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

$\mathrm{NOT}(x)$ $\neg x$

- „Logisches Nicht“

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

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

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

Logischer Operator `!`

`!b` (logisches Nicht)

`bool` $\rightarrow$ `bool`

R-Wert $\rightarrow$ R-Wert

Logischer Operator !

`!b` (logisches Nicht)

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

Logischer Operator !

`!b` (logisches Nicht)

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

Präzedenzen

```
!b && a
```

Präzedenzen

```
!b && a
```

```
(!b) && a
```

Präzedenzen

```
a && b || c && d
```

Präzedenzen

```
a && b || c && d
```

```
(a && b) || (c && d)
```

Präzedenzen

```
a || b   &&   c || d
```

Präzedenzen

```
a || b   &&   c || d
```

```
a || (b && c) || d
```

Präzedenzen

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

Präzedenzen

Der unäre logische Operator `!`
bindet stärker als

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

Präzedenzen

Der unäre logische Operator `!`
bindet stärker als
binäre arithmetische Operatoren. Diese
binden stärker als

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

Präzedenzen

Der unäre logische Operator `!`
bindet stärker als
binäre arithmetische Operatoren. Diese
binden stärker als
relationale Operatoren,
und diese binden stärker als

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

Präzedenzen

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)
```

Einige Klammern auf den vorher gezeigten Folien waren unnötig.

Vollständigkeit

- AND, OR und NOT sind die in C++ verfügbaren Booleschen Funktionen.

Vollständigkeit: $\mathrm{XOR}(x, y)$

$x \oplus y$

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

$x \oplus y$

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

Vollständigkeit: $\mathrm{XOR}(x, y)$

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

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

Vollständigkeit: $\mathrm{XOR}(x, y)$

$x \oplus y$

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

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

```
(x || y) && !(x && y)
```

Vollständigkeit Beweis

- Identifiziere binäre Boolesche Funktionen mit ihrem charakteristischen Vektor.

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

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

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

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

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}

$$f_{0000} = 0.$$

`bool` vs `int`: Konversion

- **`bool`** kann überall dort verwendet werden, wo **`int`** gefordert ist

`bool` vs `int`: Konversion

- **`bool`** kann überall dort verwendet werden, wo **`int`** gefordert ist

`bool`	→	**`int`**
true	→	1
false	→	0

`bool` vs `int`: Konversion

- **`bool`** kann überall dort verwendet werden, wo **`int`** gefordert ist – und umgekehrt.

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

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

`bool` vs `int`: Konversion

- **`bool`** kann überall dort verwendet werden, wo **`int`** gefordert ist – und umgekehrt.

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

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

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

`bool` vs `int`: Konversion

- **`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`	$\rightarrow$	**`int`**
true	$\rightarrow$	1
false	$\rightarrow$	0

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

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

DeMorgansche Regeln

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

DeMorgansche Regeln

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

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

DeMorgansche Regeln

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

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

Anwendung: Entweder … Oder (XOR)

```
(x || y)      && !(x && y)
```

Anwendung: Entweder … Oder (XOR)

```
(x || y)      && !(x && y)
```

x oder y, und nicht beide

Anwendung: Entweder … Oder (XOR)

```
(x || y)      && !(x && y)
```

x oder y, und nicht beide

```
(x || y)      && (!x || !y)
```

Anwendung: Entweder … Oder (XOR)

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

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

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)
```

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
```

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)
```

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

Kurzschlussauswertung

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

Kurzschlussauswertung

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

x hat Wert 6 $\Rightarrow$

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

Kurzschlussauswertung

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

x hat Wert 6 $\Rightarrow$

```
true && z / x > y
```

Kurzschlussauswertung

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

x hat Wert 6 $\Rightarrow$

```
true && z / x > y
```

Kurzschlussauswertung

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

x hat Wert 0 $\Rightarrow$

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

Kurzschlussauswertung

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

x hat Wert 0 $\Rightarrow$

```
false && z / x > y
```

Kurzschlussauswertung

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

x hat Wert 0 $\Rightarrow$

```
false (falsch)
```

Kurzschlussauswertung

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

x hat Wert 0 $\Rightarrow$

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

$\Rightarrow$ Keine Division durch 0

4. Defensives Programmieren

Konstanten und Assertions

Fehlerquellen

- Fehler, die der Compiler findet:
  syntaktische und manche semantische Fehler

Fehlerquellen

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

Der Compiler als Freund: Konstanten

Konstanten

- sind Variablen mit unveränderbarem Wert

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

- Verwendung: `const` vor der Definition

Der Compiler als Freund: Konstanten

Konstanten

- sind Variablen mit unveränderbarem Wert

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

- Verwendung: `const` vor der Definition

Der Compiler als Freund: Konstanten

Konstanten

- sind Variablen mit unveränderbarem Wert

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

- Verwendung: **`const`** vor der Definition

Der Compiler als Freund: Konstanten

- 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 des Versprechens *„Wert ändert sich nicht“*

Der Compiler als Freund: Konstanten

- 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 des Versprechens *„Wert ändert sich nicht“*

Der Compiler als Freund: Konstanten

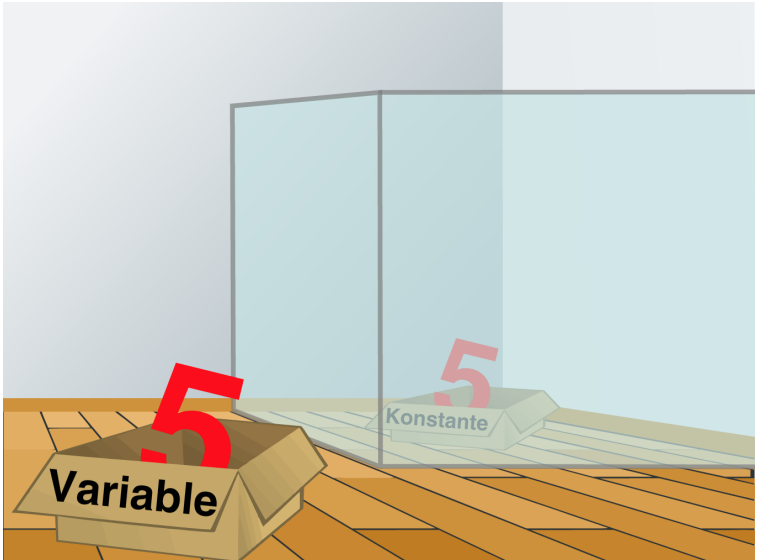
- Compiler kontrolliert Einhaltung des `const`-Versprechens

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

Compilerfehler!

- Hilfsmittel zur Vermeidung von Fehlern: Konstanten erlauben garantierte Einhaltung des Versprechens *„Wert ändert sich nicht“*

Konstanten: Variablen hinter Glas



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.

Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens

Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens

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

Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens
2. Überprüfe an vielen kritischen Stellen, ob das Programm auf dem richtigen Weg ist

Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens
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

Gegen Laufzeitfehler: *Assertions*

`assert(expr)`

- hält das Programm an, falls der boolesche Ausdruck **`expr`** nicht wahr ist

Gegen Laufzeitfehler: *Assertions*

```
assert(expr)
```

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

Gegen Laufzeitfehler: *Assertions*

`assert(expr)`

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

Assertions für den $ggT(x, y)$

Überprüfe, ob das Programm auf dem richtigen Weg ist …

```
// Input x and y
std::cout << "x =? ";
std::cin >> x;
std::cout << "y =? ";
std::cin >> y;

// Check validity of inputs
assert(x > 0 && y > 0);

... // Compute gcd(x,y), store result in variable a
```

Eingabe der Argumente für die Berechnung

Assertions für den $ggT(x, y)$

Überprüfe, ob das Programm auf dem richtigen Weg ist …

```
// Input x and y
std::cout << "x =? ";
std::cin >> x;
std::cout << "y =? ";
std::cin >> y;

// Check validity of inputs
assert(x > 0 && y > 0);
```

⟵ Vorbedingung für die weitere Berechnung

```
... // Compute gcd(x,y), store result in variable a
```

Assertions für den $ggT(x, y)$

… und hinterfrage das Offensichtliche! …

```
...
assert(x > 0 && y > 0);
```

← Vorbedingung für die weitere Berechnung

```
... // Compute gcd(x,y), store result in variable a

assert (a >= 1);
assert (x % a == 0 && y % a == 0);
for (int i = a+1; i <= x && i <= y; ++i)
  assert(!(x % i == 0 && y % i == 0));
```

Assertions für den $ggT(x, y)$

… und hinterfrage das Offensichtliche! …

```
...
assert(x > 0 && y > 0);

... // Compute gcd(x,y), store result in variable a

assert (a >= 1);
assert (x % a == 0 && y % a == 0);
for (int i = a+1; i <= x && i <= y; ++i)
  assert(!(x % i == 0 && y % i == 0));
```

Verschiedene Eigenschaften des ggT überprüfen

Assertions abschalten

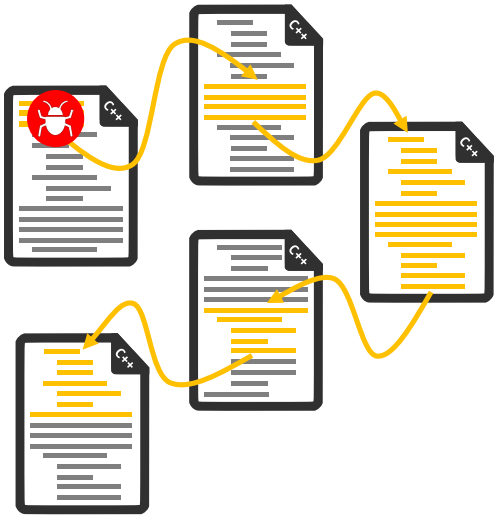
```
#define NDEBUG // To ignore assertions
#include<cassert>

...
assert(x > 0 && y > 0); // Ignored

... // Compute gcd(x,y), store result in variable a

assert(a >= 1); // Ignored
...
```

Fail-Fast mit Assertions

- Reale Software: viele C++-Dateien, komplexer Kontrollfluss

Fail-Fast mit Assertions

- Reale Software: viele C++-Dateien, komplexer Kontrollfluss



Fail-Fast mit Assertions

- Reale Software: viele C++-Dateien, komplexer Kontrollfluss
- Fehler machen sich erst spät(er) bemerkbar → Fehlersuche erschwert

Fail-Fast mit Assertions

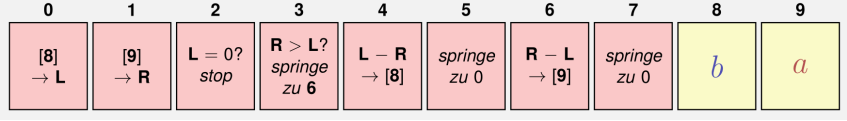
- Reale Software: viele C++-Dateien, komplexer Kontrollfluss
- Fehler machen sich erst spät(er) bemerkbar → Fehlersuche erschwert
- Assertions: Fehler frühzeitig bemerken

5. Kontrollanweisungen I

Auswahlanweisungen, Iterationsanweisungen, Terminierung, Blöcke

Kontrollfluss

- Bisher: *linear* (von oben nach unten)
- Interessante Programme nutzen „Verzweigungen“ und „Sprünge“



Auswahlanweisungen

realisieren Verzweigungen

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

`if`-Anweisung

`if` (*condition*)
    statement

if-Anweisung

if (*condition*)
    statement

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

`if`-Anweisung

`if` (*condition*)
statement

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

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

`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
```

if-else-Anweisung

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

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

if-else-Anweisung

if (*condition*)
    statement1
else
    statement2

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

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

`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.

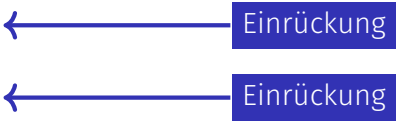
- *condition*: konvertierbar nach **`bool`**.
- *statement1*: *Rumpf* des **`if`**-Zweiges
- *statement2*: *Rumpf* des **`else`**-Zweiges

Layout!

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

Layout!

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



Iterationsanweisungen

realisieren Schleifen:

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

Berechne $1 + 2 + \ldots + n$

```
// 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";
```

Berechne $1 + 2 + \ldots + n$

```
// 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";
```

`for`-Anweisung am Beispiel

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

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

`i`	`s`

`for`-Anweisung am Beispiel

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

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

`i`	`s`
`i==1`	

`for`-Anweisung am Beispiel

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

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

`i`		`s`
`i==1`	`i <= 2`?	

`for`-Anweisung am Beispiel

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

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

`i`		`s`
`i==1`	wahr	

`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`

`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`		

`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`	`i <= 2`?	

`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	

`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`

`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`		

`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`	`i <= 2`?	

`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	

`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*)
    body statement

`for`-Anweisung: Syntax

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

- *init statement*: Ausdrucksanweisung, Deklarationsanweisung, Nullanweisung

`for`-Anweisung: Syntax

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

- *init statement*: Ausdrucksanweisung, Deklarationsanweisung, Nullanweisung
- *condition*: konvertierbar nach **`bool`**

`for`-Anweisung: Syntax

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

- *init statement*: Ausdrucksanweisung, Deklarationsanweisung, Nullanweisung
- *condition*: konvertierbar nach **`bool`**
- *expression*: beliebiger Ausdruck

`for`-Anweisung: Syntax

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

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

`for`-Anweisung: Terminierung

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

Hier und meistens:

- *expression* ändert einen Wert, der in *condition* vorkommt.

`for`-Anweisung: Terminierung

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

Hier und meistens:

- 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.

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 (init; cond; expr) stmt;
```

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.

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

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.[3]

[3] 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.

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);
```

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);
```

(Rumpf ist die Null-Anweisung)

Primzahltest: Terminierung

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

- Fortschritt: Startwert **d=2**, dann in jeder Iteration plus 1 (**++d**)

Primzahltest: Terminierung

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

- Fortschritt: Startwert `d=2`, dann in jeder Iteration plus 1 (`++d`)
- Abbruch: `n%d != 0` evaluiert zu `false` sobald ein Teiler erreicht wurde — spätestes, wenn `d == n`

Primzahltest: Terminierung

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

- Fortschritt: Startwert **`d=2`**, dann in jeder Iteration plus 1 (**`++d`**)
- Abbruch: **`n%d != 0`** evaluiert zu **`false`** sobald ein Teiler erreicht wurde — spätestes, wenn **`d == n`**
- Fortschritt garantiert, dass Abbruchbedingung erreicht wird

Primzahltest: Korrektheit

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

Jeder mögliche Teiler `2 <= d <= n` wird ausprobiert. Falls die Schleife mit `d == n` terminiert, dann und genau dann ist `n` prim.

Blöcke

- Blöcke gruppieren mehrere Anweisungen zu einer neuen Anweisung

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

Blöcke

- Blöcke gruppieren mehrere Anweisungen zu einer neuen Anweisung
- Beispiel: Rumpf der main Funktion

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

Blöcke

- Blöcke gruppieren mehrere Anweisungen zu einer neuen Anweisung
- Beispiel: Schleifenrumpf

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

Blöcke

- Blöcke gruppieren mehrere Anweisungen zu einer neuen Anweisung
- Beispiel: if / else

```
if (d < n) // d is a divisor of n in {2,...,n-1}
    std::cout << n << " = " << d << " * " << n / d << ".\n";
else {
    assert (d == n);
    std::cout << n << " is prime.\n";
}
```