3. Logical Values

Boolean Functions; the Type **`bool`**; logical and relational operators; shortcut evaluation

Our Goal

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

Our Goal

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

Behavior depends on the value of a *Boolean expression*

Our Goal

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

Behavior depends on the value of a **Boolean expression**

Boolean Values in Mathematics

Boolean expressions can take on one of two values:

0 or **1**

Boolean Values in Mathematics

Boolean expressions can take on one of two values:

0 or **1**

- **0** corresponds to **“false”**
- **1** corresponds to **“true”**

The Type `bool` in C++

- represents **logical values**

The Type `bool` in C++

- represents **logical values**
- Literals `false` and `true`

The Type `bool` in C++

- represents **logical values**
- Literals `false` and `true`
- Domain {***false***, ***true***}

```
bool b = true; // Variable with value true
```

Relational Operators

`a < b` (smaller than)

arithmetic type $\times$ arithmetic type $\rightarrow$ `bool`

R-value $\times$ R-value $\rightarrow$ R-value

Relational Operators

`a < b` (smaller than)

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

Relational Operators

`a < b` (smaller than)

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

Relational Operators

`a >= b` (greater than)

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

Relational Operators

`a >= b` (greater than)

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

Relational Operators

`a == b` (equals)

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

Relational Operators

`a == b` (equals)

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

Relational Operators

`a != b` (not equal)

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

Relational Operators

`a != b` (not equal)

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

Boolean Functions in Mathematics

- Boolean function

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

- 0 corresponds to “false”.
- 1 corresponds to “true”.

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

- "logical And"

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

- 0 corresponds to "false".
- 1 corresponds to "true".

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

Logical Operator &&

`a && b` (logical and)

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

Logical Operator `&&`

`a && b` (logical and)

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

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

- "logical Or"

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

- 0 corresponds to "false".
- 1 corresponds to "true".

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

Logical Operator ||

`a || b` (logical or)

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

Logical Operator ||

`a || b` (logical or)

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

NOT(x) $\neg x$

- "logical Not"

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

- 0 corresponds to "false".
- 1corresponds to "true".

x	NOT(x)
0	1
1	0

Logical Operator `!`

`!b` (logical not)

`bool` $\to$ **`bool`**

R-value $\to$ R-value

Logical Operator `!`

`!b` (logical not)

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

Logical Operator `!`

`!b` (logical not)

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

Precedences

```
!b && a
```

Precedences

```
!b && a
```

$$\Updownarrow$$

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

Precedences

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

Precedences

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

$$\Updownarrow$$

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

Precedences

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

Precedences

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

$$\Updownarrow$$

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

Precedences

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

Precedences

The unary logical operator `!`

binds more strongly than

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

Precedences

The unary logical operator `!`
binds more strongly than
binary arithmetic operators. These
bind more strongly than

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

Precedences

The unary logical operator `!`
binds more strongly than
binary arithmetic operators. These
bind more strongly than
relational operators,
and these bind more strongly than

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

Precedences

The unary logical operator `!`
binds more strongly than
binary arithmetic operators. These
bind more strongly than
relational operators,
and these bind more strongly than
binary logical operators.

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

Some parentheses on the previous slides were actually redundant.

Completeness

- AND, OR and NOT are the boolean functions available in C++.

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

- AND, OR and NOT are the boolean functions available in $\mathrm{C}{++}$.
- Any other *binary* boolean function can be generated from them.

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

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

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

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

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

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

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

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

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

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

Completeness Proof

- Identify binary boolean functions with their characteristic vector.

Completeness Proof

- Identify binary boolean functions with their characteristic vector.

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

Completeness Proof

- Identify binary boolean functions with their characteristic vector.

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

characteristic vector: 0110

Completeness Proof

- Identify binary boolean functions with their characteristic vector.

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

characteristic vector: 0110

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

Completeness Proof

- Step 1: generate the *fundamental* functions f_{0001}, f_{0010}, f_{0100}, f_{1000}

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

Completeness Proof

- Step 2: generate all functions by applying logical or

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

Completeness Proof

- Step 2: generate all functions by applying logical or

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

- Step 3: generate f_{0000}

$$f_{0000} = 0.$$

`bool` vs `int`: Conversion

- `bool` can be used whenever `int` is expected

`bool` vs `int`: Conversion

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

- **`bool`** can be used whenever **`int`** is expected

`bool` vs `int`: Conversion

- `bool` can be used whenever `int` is expected – and vice versa.

`bool`	→	`int`
true	→	1
false	→	0

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

`bool` vs `int`: Conversion

- `bool` can be used whenever `int` is expected – and vice versa.

`bool`	→	`int`
true	→	1
false	→	0

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

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

`bool` vs `int`: Conversion

- `bool` can be used whenever `int` is expected – and vice versa.
- Many existing programs use `int` instead of `bool`

  This is bad style originating from the language C .

`bool`	→	`int`
true	→	1
false	→	0

`int`	→	`bool`
≠0	→	*true*
0	→	*false*

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

DeMorgan Rules

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

DeMorgan Rules

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

! (rich *and* beautiful) == (poor *or* ugly)

DeMorgan Rules

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

! (rich *and* beautiful) == (poor *or* ugly)

Application: either … or (XOR)

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

Application: either ... or (XOR)

`(x || y)      && !(x && y)` x or y, and not both

Application: either ... or (XOR)

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

x or y, and not both

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

Application: either ... or (XOR)

`(x || y)      && !(x && y)` x or y, and not both

`(x || y)      && (!x || !y)` x or y, and one of them not

Application: either ... or (XOR)

```
(x || y)      && !(x && y)
```
x or y, and not both

```
(x || y)      && (!x || !y)
```
x or y, and one of them not

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

Application: either ... or (XOR)

```
(x || y)    && !(x && y)
```
x or y, and not both

```
(x || y)    && (!x || !y)
```
x or y, and one of them not

```
!(!x && !y) && !(x && y)
```
not none and not both

Application: either ... or (XOR)

```
(x || y)     && !(x && y)
```
x or y, and not both

```
(x || y)     && (!x || !y)
```
x or y, and one of them not

```
!(!x && !y)  && !(x && y)
```
not none and not both

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

Application: either ... or (XOR)

`(x || y)`	`&& !(x && y)`	x or y, and not both
`(x || y)`	`&& (!x || !y)`	x or y, and one of them not
`!(!x && !y)`	`&& !(x && y)`	not none and not both
`!(!x && !y || x && y)`		not: both or none

Short circuit Evaluation

- Logical operators `&&` and `||` evaluate the *left operand first*.
- If the result is then known, the right operand will *not be* evaluated.

Short circuit Evaluation

- Logical operators `&&` and `||` evaluate the *left operand first.*
- If the result is then known, the right operand will *not be* evaluated.

x has value 6 $\Rightarrow$

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

Short circuit Evaluation

- Logical operators **`&&`** and `||` evaluate the *left operand first.*
- If the result is then known, the right operand will *not be* evaluated.

x has value 6 $\Rightarrow$

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

Short circuit Evaluation

- Logical operators **&&** and **||** evaluate the *left operand first.*
- If the result is then known, the right operand will *not be* evaluated.

x has value 6 $\Rightarrow$

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

Short circuit Evaluation

- Logical operators **&&** and **||** evaluate the *left operand first*.
- If the result is then known, the right operand will *not be* evaluated.

x has value 0 $\Rightarrow$

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

Short circuit Evaluation

- Logical operators `&&` and `||` evaluate the *left operand first*.
- If the result is then known, the right operand will *not be* evaluated.

x has value 0 $\Rightarrow$

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

Short circuit Evaluation

- Logical operators `&&` and `||` evaluate the *left operand first*.
- If the result is then known, the right operand will *not be* evaluated.

x has value 0 $\Rightarrow$

```
false
```

Short circuit Evaluation

- Logical operators **&&** and **||** evaluate the *left operand first*.
- If the result is then known, the right operand will *not be* evaluated.

x has value 0 $\Rightarrow$

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

$\Rightarrow$ No division by 0

4. Defensive Programming

Constants and Assertions

Sources of Errors

- Errors that the compiler can find:
  syntactical and some semantical errors

Sources of Errors

- Errors that the compiler can find:
  syntactical and some semantical errors
- Errors that the compiler cannot find:
  runtime errors (always semantical)

The Compiler as Your Friend: Constants

Constants

- are variables with immutable value

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

- Usage: `const` before the definition

The Compiler as Your Friend: Constants

Constants

- are variables with immutable value

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

- Usage: `const` before the definition

The Compiler as Your Friend: Constants

Constants

- are variables with immutable value

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

- Usage: **`const`** before the definition

The Compiler as Your Friend: Constants

- Compiler checks that the **const**-promise is kept

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

compiler: error

- Tool to avoid errors: constants guarantee the promise :*“value does not change”*

The Compiler as Your Friend: Constants

- Compiler checks that the **const**-promise is kept

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

compiler: error

- Tool to avoid errors: constants guarantee the promise :*“value does not change”*

The Compiler as Your Friend: Constants

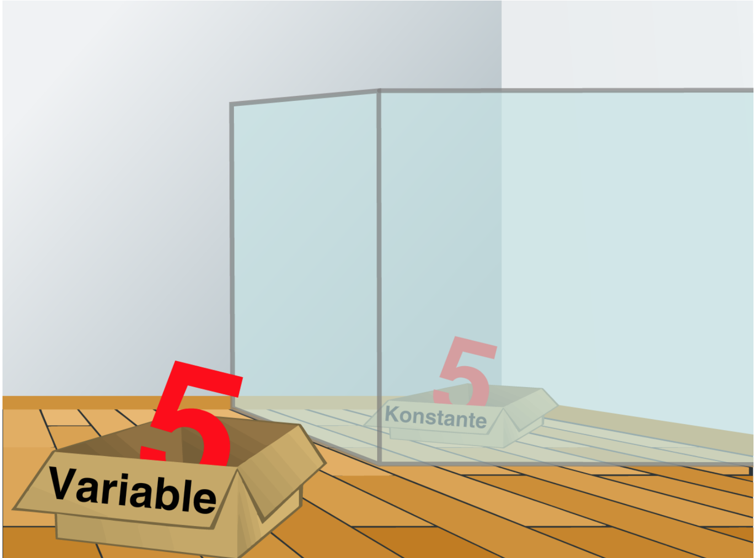
- Compiler checks that the **`const`**-promise is kept

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

compiler: error

- Tool to avoid errors: constants guarantee the promise :“*value does not change*”

Constants: Variables behind Glass



The `const`-guideline

const-guideline

For *each variable*, think about whether it will change its value in the lifetime of a program. If not, use the keyword **`const`** in order to make the variable a constant.

A program that adheres to this guideline is called **`const`**-correct.

Avoid Sources of Bugs

1. Exact knowledge of the wanted program behavior

Avoid Sources of Bugs

1. Exact knowledge of the wanted program behavior

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

Avoid Sources of Bugs

1. Exact knowledge of the wanted program behavior
2. Check at many places in the code if the program is still on track

Avoid Sources of Bugs

1. Exact knowledge of the wanted program behavior
2. Check at many places in the code if the program is still on track
3. Question the (seemingly) obvious, there could be a typo in the code

Against Runtime Errors: *Assertions*

```
assert(expr)
```

- halts the program if the boolean expression **`expr`** is false

Against Runtime Errors: *Assertions*

```
assert(expr)
```

- halts the program if the boolean expression **`expr`** is false
- requires **`#include <cassert>`**

Against Runtime Errors: *Assertions*

```
assert(expr)
```

- halts the program if the boolean expression **`expr`** is false
- requires **`#include <cassert>`**
- can be switched off (potential performance gain)

Assertions for the $gcd(x, y)$

Check if the program is on track . . .

```
// 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
```

Input arguments for calculation

Assertions for the $gcd(x, y)$

Check if the program is on track ...

```
// 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
```

← Precondition for the ongoing computation

Assertions for the $gcd(x, y)$

... and question the obvious! ...

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

← Precondition for the ongoing computation

```
... // 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 for the $gcd(x, y)$

... and question the obvious! ...

```
...
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));
```

Properties of the gcd

Switch off Assertions

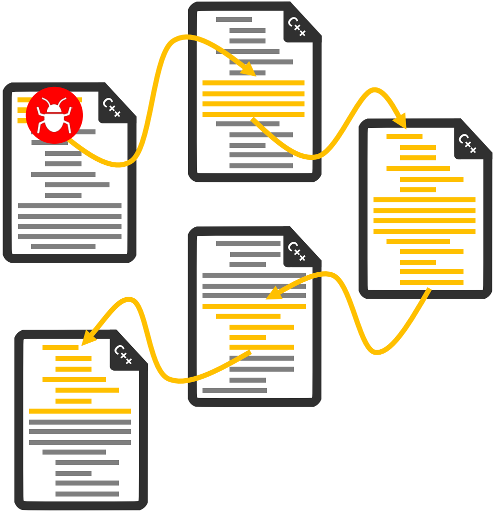
```
#define NDEBUG // To ignore assertions
#include<cassert>

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

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

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

Fail-Fast with Assertions

- Real software: many C++ files, complex control flow

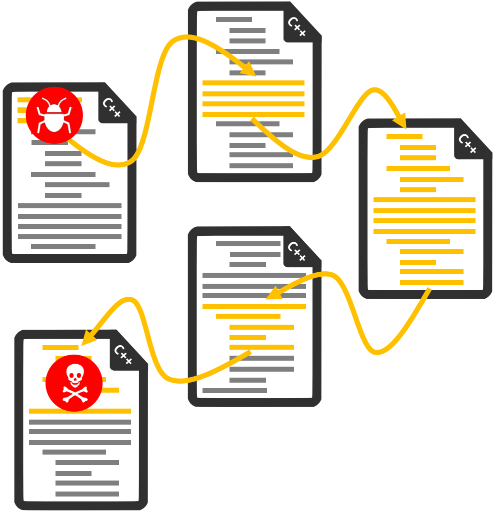
Fail-Fast with Assertions

- Real software: many C++ files, complex control flow



Fail-Fast with Assertions

- Real software: many C++ files, complex control flow
- Errors surface late(r) $\rightarrow$ impedes error localisation



Fail-Fast with Assertions

- Real software: many C++ files, complex control flow
- Errors surface late(r) $\rightarrow$ impedes error localisation
- Assertions: Detect errors early

5. Control Structures I

Selection Statements, Iteration Statements, Termination, Blocks

Control Flow

- Up to now: *linear* (from top to bottom)
- Interesting programs require “branches” and “jumps”

```
// Project Hangman
...
while (game_not_over) {
  ...
  if (word.contains(guess)) {
    ...
  } else {
    ...
  }
}
...
```

Selection Statements

implement branches

- `if` statement
- `if-else` statement

`if`-Statement

`if` (*condition*)
    statement

`if`-Statement

`if` (*condition*)
statement

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

`if`-Statement

`if` (*condition*)
    statement

If *condition* is true then *statement* is executed

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

`if`-Statement

`if` (*condition*)
  statement

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

If *condition* is true then *statement* is executed

- *statement*: arbitrary statement (*body* of the **`if`**-Statement)
- *condition*: convertible to **`bool`**

`if-else`-statement

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

`if-else`-statement

if (*condition*)
    statement1
else
    statement2

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

if-else-statement

if (*condition*)
    statement1
else
    statement2

If *condition* is true then *statement1* is executed, otherwise *statement2* is executed.

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

`if-else`-statement

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

If *condition* is true then *statement1* is executed, otherwise *statement2* is executed.

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

- *condition*: convertible to **`bool`**.
- *statement1*: *body* of the **`if`**-branch
- *statement2*: *body* of the **`else`**-branch

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";   ← Indentation
else
    std::cout << "odd";    ← Indentation
```

Iteration Statements

implement loops

- `for`-statement
- `while`-statement
- `do`-statement

Compute $1 + 2 + ... + n$

```
// 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";
```

Compute $1 + 2 + ... + n$

```
// 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`-Statement Example

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

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

i	s

for-Statement Example

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

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

i	s
i==1	

for-Statement Example

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

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

i		s
i==1	i <= 2?	

for-Statement Example

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

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

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

for-Statement Example

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

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

i		s
i==1	wahr	s == 1

for-Statement Example

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

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

i		s
i==1	wahr	s == 1
i==2		

for-Statement Example

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

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

i		s
i==1	wahr	s == 1
i==2	i <= 2?	

`for`-Statement Example

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

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

`i`		`s`
`i==1`	wahr	`s == 1`
`i==2`	wahr	

for-Statement Example

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

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

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

`for`-Statement Example

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

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

`i`		`s`
`i==1`	wahr	`s == 1`
`i==2`	wahr	`s == 3`
`i==3`		

`for`-Statement Example

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

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

`i`		`s`
`i==1`	wahr	`s == 1`
`i==2`	wahr	`s == 3`
`i==3`	`i <= 2`?	

for-Statement Example

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

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

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

for-Statement Example

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

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

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

s == 3

`for`-Statement: Syntax

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

`for`-Statement: Syntax

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

- *init statement*: expression statement, declaration statement, null statement

`for`-Statement: Syntax

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

- *init statement*: expression statement, declaration statement, null statement
- *condition*: convertible to **`bool`**

`for`-Statement: Syntax

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

- *init statement*: expression statement, declaration statement, null statement
- *condition*: convertible to `bool`
- *expression*: any expression

`for`-Statement: Syntax

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

- *init statement*: expression statement, declaration statement, null statement
- *condition*: convertible to **`bool`**
- *expression*: any expression
- *body statement*: any statement (*body* of the for-statement)

for-Statement: Termination

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

Here and in most cases:

- *expression* changes its value that appears in *condition* .

for-Statement: Termination

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

Here and in most cases:

- After a finite number of iterations *condition* becomes false: **Termination**

Infinite Loops

- Infinite loops are easy to generate:

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

  - Die *empty condition* is true.
  - Die *empty expression* has no effect.
  - Die *null statement* has no effect.

Infinite Loops

- Infinite loops are easy to generate:

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

  - Die *empty condition* is true.
  - Die *empty expression* has no effect.
  - Die *null statement* has no effect.

- … but can in general not be automatically detected.

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

Halting Problem

Undecidability of the Halting Problem

There is no C++ program that can determine for each C++-Program P and each input I if the program P terminates with the input I.

[4] Alan Turing, 1936. Theoretical questions of this kind were the main motivation for Alan Turing to construct a computing machine.

Halting Problem

Undecidability of the Halting Problem

There is no $\mathrm{C}{+}{+}$ program that can determine for each $\mathrm{C}{+}{+}$-Program P and each input I if the program P terminates with the input I.

This means that the correctness of programs can in general *not* be automatically checked.[4]

[4] Alan Turing, 1936. Theoretical questions of this kind were the main motivation for Alan Turing to construct a computing machine.

Example: Prime Number Test

Def.: a natural number $n \geq 2$ is a prime number, if no $d \in \{2, \ldots, n-1\}$ divides n .

Example: Prime Number Test

Def.: a natural number $n \geq 2$ is a prime number, if no $d \in \{2, \ldots, n-1\}$ divides n .

A loop that can test this:

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

Example: Prime Number Test

Def.: a natural number $n \geq 2$ is a prime number, if no $d \in \{2, \ldots, n-1\}$ divides n .

A loop that can test this:

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

(body is the null statement)

Example: Termination

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

- Progress: Initial value **`d=2`**, then plus 1 in every iteration (**`++d`**)

Example: Termination

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

- Progress: Initial value `d=2`, then plus 1 in every iteration (`++d`)
- Exit: `n%d != 0` evaluates to `false` as soon as a divisor is found — at the latest, once `d == n`

Example: Termination

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

- Progress: Initial value **`d=2`**, then plus 1 in every iteration (**`++d`**)
- Exit: **`n%d != 0`** evaluates to **`false`** as soon as a divisor is found — at the latest, once **`d == n`**
- Progress guarantees that the exit condition will be reached

Example: Correctness

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

Every potential divisor **`2 <= d <= n`** will be tested. If the loop terminates with **`d == n`** then and only then is **`n`** prime.

Blocks

- Blocks group a number of statements to a new statement

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

Blocks

- Blocks group a number of statements to a new statement
- Example: body of the main function

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

Blocks

- Blocks group a number of statements to a new statement
- Example: loop body

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

Blocks

- Blocks group a number of statements to a new statement
- 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";
}
```