# 2. Logical Values

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



## 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*

- *0* corresponds to *“false”*
- *1* corresponds to *“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)
a >= b    (greater than)
a == b    (equals)
a != b    (not equal)
```

arithmetic type $\times$ arithmetic type $\to$ `bool`

R-value $\times$ R-value $\to$ R-value



## Table of Relational Operators

| | Symbol | Arity | Precedence | Associativity |
|---|---|---|---|---|
| **smaller** | < | 2 | 11 | left |
| **greater** | > | 2 | 11 | left |
| **smaller equal** | <= | 2 | 11 | left |
| **greater equal** | >= | 2 | 11 | left |
| **equal** | == | 2 | 10 | left |
| **unequal** | != | 2 | 10 | left |

arithmetic type $\times$ arithmetic type $\to$ `bool`

R-value $\times$ R-value $\to$ R-value



## Boolean Functions in Mathematics

- Boolean function

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

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



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

- “logical And”

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

- $0$ corresponds to “false”.
- 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)

bool $\times$ bool $\rightarrow$ bool

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

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



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

- “logical Or”

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

- 0 corresponds to “false”.
- 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)

bool $\times$ bool $\rightarrow$ bool

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

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



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

- “logical Not”

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

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

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

## Logical Operator !

`!b` (logical not)

`bool` → `bool`

R-value → R-value

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



## Precedences

```
!b && a
   ⇕
(!b) && a

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

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



## Table of Logical Operators

| | Symbol | Arity | Precedence | Associativity |
|---|---|---|---|---|
| **Logical and (AND)** | `&&` | 2 | 6 | left |
| **Logical or (OR)** | `\|\|` | 2 | 5 | left |
| **Logical not (NOT)** | `!` | 1 | 16 | right |



## 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.

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

## Completeness

- $\mathrm{AND}$, $\mathrm{OR}$ and $\mathrm{NOT}$ 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: $\mathrm{XOR}(x,y)$ $\quad x \oplus y$

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

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

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



## Completeness Proof

- Identify binary boolean functions with their characteristic vector.

| $x$ | $y$ | $\mathrm{XOR}(x,y)$ |
|---|---|---|
| 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 $f_{0001}$, $f_{0010}$, $f_{0100}$, $f_{1000}$

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

## 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 – and vice versa.
- Many existing programs use `int` instead of `bool`
  *This is bad style originating from the language $C$ .*

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

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

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



## DeMorgan Rules

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

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



## 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.

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

⇒ No division by 0



## Sources of Errors

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



## Avoid Sources of Bugs

1. Exact knowledge of the wanted program behavior

   ≫ It's not a bug, it's a feature !!≪
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
- requires `#include <cassert>`
- can be switched off

## DeMorgan's Rules

Question the obvious Question the seemingly obvious!

```
// 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;
}
```



## Switch off Assertions

```
// 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 Identity a/b * b + a%b == a

Check if the program is on track...

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

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

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

Input arguments for calculation

Precondition for the ongoing computation



## Div-Mod identity a/b * b + a%b == a

...and question the obvious!

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

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

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

Precondition for the ongoing computation

Div-Mod identity

# 3. Control Structures I

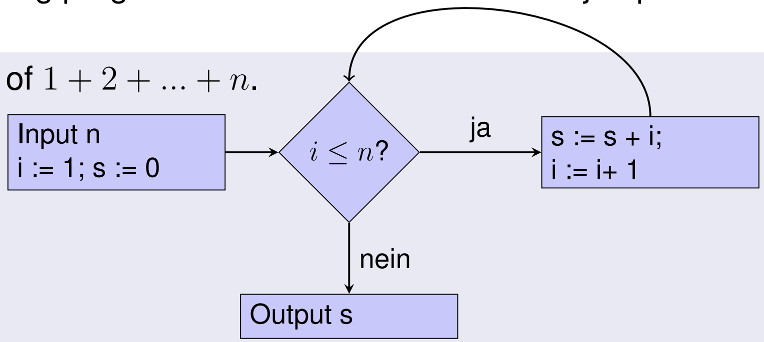
Selection Statements, Iteration Statements, Termination, Blocks



## Control Flow

- up to now *linear* (from top to bottom)
- For interesting programs we need “branches” and “jumps”

Computation of $1 + 2 + ... + n$.






## Selection Statements

implement branches

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



## `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*

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

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

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



## Iteration Statements

implement “loops”

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



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

```
// 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-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* )
    *statement*

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



## for-Statement: semantics

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

- *init-statement* is executed
- *condition* is evaluated
  - `true`: Iteration starts
    *statement* is executed
    *expression* is executed
  - false: `for`-statement is ended.



## Gauß as a Child (1777 - 1855)

- Math-teacher wanted to keep the pupils busy with the following task:

*Compute the sum of numbers from 1 to 100 !*

- Gauß finished after one minute.

## The Solution of Gauß

- The requested number is

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

- This is half of

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

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



## `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* .
- 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.

- ... but can in general not be automatically detected.

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



## 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.[1]

[1] Alan Turing, 1936. Theoretical quesitons 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 $n$ .

A loop that can test this:

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

- Observation 1:
  After the `for`-statement it holds that $d \leq n$.
- Observation 2:
  $n$ is a prime number if and only if finally $d = n$.



## Blocks

- Blocks group a number of statements to a new statement
  `{statement1 statement2 ... statementN}`
- Example: body of the main function

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

- Example: loop body

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