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";
```

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 "wrong"
- 1 corresponds to "true"

The Type bool in C++

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

```cpp
bool b = true; // Variable with value true
```
### Relational Operators

- \( a < b \) (smaller than)
- \( a \geq b \) (greater than)
- \( a == b \) (equals)
- \( a \neq b \) (unequal)

```
number type \times number type \rightarrow \text{bool}
R-value \times R-value \rightarrow R-value
```

### Table of Relational Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Arity</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>smaller</td>
<td>&lt;</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>greater</td>
<td>&gt;</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>smallerEqual</td>
<td>&lt;=</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>greaterEqual</td>
<td>&gt;=</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>equal</td>
<td>==</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>unequal</td>
<td>!=</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

### Boolean Functions in Mathematics

- Boolean function

\[
f : \{0, 1\}^2 \rightarrow \{0, 1\}
\]

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

### AND \((x, y)\)

- “logical and”

\[
f : \{0, 1\}^2 \rightarrow \{0, 1\}
\]

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

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
<th>AND ((x, y))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
### Logical Operator &&

a && b  
(logical and)

bool × bool → bool
R-value × R-value → R-value

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

### Logical Operator ||

a || b  
(logical or)

bool × bool → bool
R-value × R-value → R-value

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

### OR(x, y)  
\[ x \lor y \]

- “logical or”
- \( f : \{0, 1\}^2 \rightarrow \{0, 1\} \)
- 0 corresponds to “false”.
- 1 corresponds to “true”.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>OR(x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### NOT(x)  
\[ \neg x \]

- “logical not”
- \( f : \{0, 1\} \rightarrow \{0, 1\} \)
- 0 corresponds to “false”.
- 1 corresponds to “true”.

<table>
<thead>
<tr>
<th>x</th>
<th>NOT(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Logical Operator !

!b  (logical not)

bool → bool
R-value → R-value

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

The unary logical operator ! provides a stronger binding than
binary arithmetic operators. These
bind stronger than
relational operators,
and these bind stronger than
binary logical operators.

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

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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>XOR(x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Completeness: XOR(x, y)

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

\[ x \oplus y = (x \lor y) \land \neg(x \land y). \]

\[ (x || y) \&\& ! (x \&\& y) \]

Completeness Proof

- Identify binary boolean functions with their characteristic vector.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>XOR(x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

characteristic vector: 0110

XOR = \(f_{0110}\)

Completeness Proof

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

\[ f_{0001} = \text{AND}(x, y) \]
\[ f_{0010} = \text{AND}(x, \text{NOT}(y)) \]
\[ f_{0100} = \text{AND}(y, \text{NOT}(x)) \]
\[ f_{1000} = \text{NOT}(\text{OR}(x, y)) \]
Completeness Proof

- Step 2: generate all functions by applying logical or
  \[ f_{1101} = \text{OR}(f_{1000}, \text{OR}(f_{0100}, f_{0001})) \]

- 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.

<table>
<thead>
<tr>
<th>bool</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>false</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int</th>
<th>bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>≠0</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>false</td>
</tr>
</tbody>
</table>

bool b = 3; // b=true

DeMorgan Rules

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

Application: either ... or (XOR)

- \( (x \lor y) \&\& !(x \&\& y) \) x or y, and not both
- \( (x \lor y) \&\& (!x \lor !y) \) 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
Shortcut Evaluation

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

\[ x \neq 0 \land z / x > y \Rightarrow \text{No division by 0} \]

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
   \[ \Rightarrow \text{It's not a bug, it's a feature} \]
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

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);

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

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

Div-Mod identity

a/b * b + a%b == a

Precondition for the ongoing computation

Input arguments for calculation

Precondition for the ongoing computation

Precondition for the ongoing computation
4. 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 $1 + 2 + \ldots + n$.

Eingabe $n$

\[ i := 1; \ s := 0 \]

\[ i \leq n? \]

ja

\[ s := s + i; \]
\[ i := i + 1 \]

nein

Ausgabe $s$

Selection Statements

implement branches

- if statement
- if-else statement

if-Statement

\[
\text{if ( condition )} \\
\text{statement}
\]

int $a$;
std::cin >> $a$;
if (a % 2 == 0)
    std::cout << "even";
if-else-statement

if (condition)
statement1
else
statement2

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

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";

Indentation

Iteration Statements

Implement "loops"

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

Compute $1 + 2 + \ldots + n$

// 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-Statement Example**

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

**Assumptions:**
- $n = 2$, $s = 0$
  - $i = 1$: $s = 1$
  - $i = 2$: $s = 3$
  - $i = 3$: $s = 3$

---

**for-Statement: Syntax**

```c
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**

```c
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
  \[ 1 + 2 + 3 + \cdots + 98 + 99 + 100. \]

- being half of
  \[
  1 + 2 + \cdots + 99 + 100 \\
  + 100 + 99 + \cdots + 2 + 1 \\
  = 101 + 101 + \cdots + 2 + 1 \\
  \]

- Answer: \( 100 \cdot \frac{101}{2} = 5050 \)

for-Statement: Termination

\[
\text{for (unsigned int } i = 1; i <= n; ++i) \\
\quad s += i; \\
\]

Hier und meistens:

- \textit{expression} changes its value that appears in \textit{condition}.
- After a finite number of iterations \textit{condition} becomes false: \textit{Termination}

Endless Loops

- Endless loops are easy to generate:
  \[
  \text{for ( ; ; ) ;} \\
  \]

  - Die \textit{empty condition} is true.
  - Die \textit{empty expression} has no effect.
  - Die \textit{null statement} has no effect.

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

  \[
  \text{for ( e; v; e) r;} \\
  \]

Halting Problem

Undecidability of the Halting Problem

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

This means that the correctness of programs can in general \textit{not} be automatically checked.\textsuperscript{5}

\textsuperscript{5}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 $n$.

A loop that can test this:

```c
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**

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

- **Example: loop body**

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