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)
- \( a \geq b \) (greater than)
- \( a == b \) (equals)
- \( a \neq b \) (not equal)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Arity</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>smaller</td>
<td>2</td>
<td>11</td>
<td>left</td>
</tr>
<tr>
<td>greater</td>
<td>2</td>
<td>11</td>
<td>left</td>
</tr>
<tr>
<td>smaller equal</td>
<td>2</td>
<td>11</td>
<td>left</td>
</tr>
<tr>
<td>greater equal</td>
<td>2</td>
<td>11</td>
<td>left</td>
</tr>
<tr>
<td>equal</td>
<td>2</td>
<td>10</td>
<td>left</td>
</tr>
<tr>
<td>unequal</td>
<td>2</td>
<td>10</td>
<td>left</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”**

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( \text{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>

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

R-value \( \times \) R-value \( \rightarrow \) R-value

arithmetic type \( \times \) arithmetic type \( \rightarrow \) bool
Logical Operator \&\&

\[
\text{a } \&\& \text{ b} \quad \text{(logical and)}
\]

\[
\text{bool } \times \text{ bool} \rightarrow \text{ bool}
\]

\[
\text{R-value } \times \text{ R-value} \rightarrow \text{ R-value}
\]

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

OR \((x, y)\)

\[
\text{x } \lor \text{ y}
\]

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

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

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

Logical Operator \|\|

\[
\text{a } \mid\!\!\!\mid \text{ b} \quad \text{(logical or)}
\]

\[
\text{bool } \times \text{ bool} \rightarrow \text{ bool}
\]

\[
\text{R-value } \times \text{ R-value} \rightarrow \text{ R-value}
\]

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

NOT \((x)\)

\[
\neg x
\]

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

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

```
```
**Logical Operator !**

\[ !b \quad \text{(logical not)} \]

\[ \text{bool} \rightarrow \text{bool} \]
\[ \text{R-value} \rightarrow \text{R-value} \]

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

---

**Precedences**

\[ !b \land a \]
\[ (!b) \land a \]

\[ a \land b \lor c \land d \]
\[ (a \land b) \lor (c \land d) \]

\[ a \lor b \land c \lor d \]
\[ a \lor (b \land c) \lor d \]

---

**Table of Logical Operators**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Arity</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical and (AND)</td>
<td>&amp;&amp;</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Logical or (OR)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical not (NOT)</td>
<td>!</td>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

---

**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.

```c
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) $x \oplus y$

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

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

\[
(x \lor y) \land \neg(x \land y)
\]

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

\[
\text{XOR} = f_{0110}
\]

Completeness Proof

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

\[
f_{0001} = \text{AND}(x, y)
\]

\[
f_{0010} = \text{AND}(x, \neg y)
\]

\[
f_{0100} = \text{AND}(y, \neg x)
\]

\[
f_{1000} = \neg(\text{OR}(x, y))
\]
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 → int</th>
<th>int → bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>true → 1</td>
<td>≠0 → true</td>
</tr>
<tr>
<td>false → 0</td>
<td>0 → false</td>
</tr>
</tbody>
</table>

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.

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

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!

```c++
// 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

```c++
// 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);
```

...and question the obvious!

```
// check input
assert (b != 0); Precondition for the ongoing computation

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

// check result
assert (div * b + mod == a); 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 $1 + 2 + \ldots + n$.

\[
\begin{align*}
\text{Input } & n \\
i & := 1; s := 0 \\
i & \leq n? \\
\text{ja} & \\
s & := s + i; \\
i & := i + 1 \\
\text{nein} & \\
\text{Output } & s
\end{align*}
\]

Selection Statements

implement branches
- if statement
- if-else statement

if-Statement

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

If *condition* is true then *statement* is executed
- *statement*: arbitrary statement (*body of the if-Statement*)
- *condition*: convertible to *bool*

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

```c++
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

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

**Iteration Statements**

Implement "loops"

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

```c++
// 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;
}
```

**Compute** \(1 + 2 + \ldots + n\)

```c++
int a;
std::cin >> a;
if (a % 2 == 0)  
  std::cout << "even";
else  
  std::cout << "odd";
```
for-Statement Example

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

Assumptions: n == 2, s == 0

<table>
<thead>
<tr>
<th>i</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>i==1</td>
<td>wahr</td>
</tr>
<tr>
<td>i==2</td>
<td>wahr</td>
</tr>
<tr>
<td>i==3</td>
<td>falsch</td>
</tr>
</tbody>
</table>

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
  \[
  \begin{align*}
  &1 + 2 + \cdots + 99 + 100 \\
  &+ 100 + 99 + \cdots + 2 + 1 \\
  = &\quad 101 + 101 + \cdots + 101 + 101
  \end{align*}
  \]

- Answer: \(100 \cdot 101/2 = 5050\)

for-Statement: Termination

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

Here and in most cases:

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

Infinite Loops

- Infinite 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 \textit{C++} program that can determine for each \textit{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 \textit{not} be automatically checked.¹

¹ 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";
  }
  ```