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 “false”
- 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)
- $a != b$ (not equal)

<table>
<thead>
<tr>
<th>Operation</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”.

$\text{AND}(x, y) = x \land y$

- “logical And”

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

<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>
Logical Operator \(\&\&\)

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

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

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

Logical Operator \(\|\|

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

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

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

Logical Operator \(\lor\lor\)

\[ x \lor y \]

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

\[ x y \mid OR(x, y) \]
\[ \begin{array}{ccc}
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 1 \\
  1 & 1 & 1 \\
\end{array} \]

NOT \((x)\)

\[ \neg x \]

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

\[ x \mid NOT(x) \]
\[ \begin{array}{c}
  0 \\
  1 \\
\end{array} \]
Int n = 1;
bool b = !(n < 0);  // b = true

Precedences

Logical Operator !

! b  (logical not)

bool → bool
R-value → R-value

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

Any other binary boolean function can be generated from them.

\[
\begin{array}{c|c|c}
 x & y & \text{XOR}(x, y) \\
 0 & 0 & 0 \\
 0 & 1 & 1 \\
 1 & 0 & 1 \\
 1 & 1 & 0 \\
\end{array}
\]

**Completeness Proof**

Identify binary boolean functions with their characteristic vector.

\[
\begin{array}{c|c|c}
 x & y & \text{XOR}(x, y) \\
 0 & 0 & 0 \\
 0 & 1 & 1 \\
 1 & 0 & 1 \\
 1 & 1 & 0 \\
\end{array}
\]

characteristic vector: 0110

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

**Completeness: XOR(\(x, y\))**

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

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

\[
(x \mid\mid y) \&\& \neg(x \&\& y)
\]

**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 → 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.

\[ x \neq 0 \land z / x > y \]

⇒ No division by 0

4. Defensive Programming

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
  
  \[ \text{const int speed_of_light} = 299792458; \]

- Usage: \texttt{const} before the definition
The Compiler as Your Friend: Constants

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

```c
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

- 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
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**

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

```cpp
assert(expr)
```

**Assertions for the `gcd(x, y)`**

... and question the obvious! ...

```cpp
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));
```

**Switch off Assertions**

```cpp
#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
- Errors surface late(r) → 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”

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

Selection Statements

- implement branches
  - if statement
  - if-else statement
If `condition` is true then `statement` is executed.

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

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";
```
Compute $1 + 2 + \ldots + n$

```cpp
// 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;
}
```

### Gauß as a Child (1777 - 1855)

- As you probably know, there exists a more efficient way to compute the sum of the first $n$ natural numbers. Here’s a corresponding anecdote:
- 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
  
  $$\begin{array}{c}
  1 + 2 + \cdots + 99 + 100 \\
  + 100 + 99 + \cdots + 2 + 1 \\
  \end{array}
  = \begin{array}{c}
  101 + 101 + \cdots + 101 + 101 \\
  \end{array}$$

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

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 (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 $I$.

This means that the correctness of programs can in general not be automatically checked.\(^4\)

---

**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);
```

---

**Example: Termination**

```c++
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:** $n%d \neq 0$ 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**

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

Every potential divisor $2 \leq d \leq n$ 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}`

- 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";
  }
  ```