

3. 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**

143

144

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
```

145

146

Relational Operators

$a < b$ (smaller than)
 $a >= b$ (greater than)
 $a == b$ (equals)
 $a != b$ (not equal)

arithmetic type \times arithmetic type \rightarrow bool

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

147

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 \rightarrow bool

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

148

Boolean Functions in Mathematics

- Boolean function

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

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

149

AND(x, y)

$$x \wedge y$$

- "logical And"

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

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

x	y	AND(x, y)
0	0	0
0	1	0
1	0	0
1	1	1

150

Logical Operator &&

`a && b` (logical and)

`bool × bool → bool`

`R-value × R-value → R-value`

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

151

OR(x, y)

$x \vee y$

- “logical Or”

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

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

x	y	OR(x, y)
0	0	0
0	1	1
1	0	1
1	1	1

152

Logical Operator ||

`a || b` (logical or)

`bool × bool → bool`

`R-value × R-value → R-value`

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

153

NOT(x)

$\neg x$

- “logical Not”

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

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

x	NOT(x)
0	1
1	0

154

Logical Operator !

!b (logical not)

bool → bool

R-value → R-value

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

155

Precedences

$$\begin{array}{c} !b \ \&\& \ a \\ \Updownarrow \\ (!b) \ \&\& \ a \\ \\ a \ \&\& \ b \ || \ c \ \&\& \ d \\ \Updownarrow \\ (a \ \&\& \ b) \ || \ (c \ \&\& \ d) \\ \\ a \ || \ b \ \ \ \ \ \&\& \ \ \ \ \ \ c \ || \ d \\ \Updownarrow \\ a \ || \ (b \ \&\& \ c) \ || \ d \end{array}$$

156

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

157

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

158

Completeness

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

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

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

$$x \oplus y$$

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

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

$$(x \ || \ y) \ \&\& \ \!(x \ \&\& \ y)$$

159

160

Completeness Proof

- Identify binary boolean functions with their characteristic vector.

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

characteristic vector: 0110

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

Completeness Proof

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

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

$$f_{0010} = \text{AND}(x, \text{NOT}(y))$$

$$f_{0100} = \text{AND}(y, \text{NOT}(x))$$

$$f_{1000} = \text{NOT}(\text{OR}(x, y))$$

161

162

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}

$$f_{0000} = 0.$$

163

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.

<code>bool</code>	→	<code>int</code>
<code>true</code>	→	1
<code>false</code>	→	0
<code>int</code>	→	<code>bool</code>
<code>≠0</code>	→	<code>true</code>
0	→	<code>false</code>

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

164

DeMorgan Rules

- $\neg(a \ \&\& \ b) \equiv (\neg a \ || \ \neg b)$
- $\neg(a \ || \ b) \equiv (\neg a \ \&\& \ \neg b)$

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

165

Application: either ... or (XOR)

$(x \ || \ y) \ \&\& \ \neg(x \ \&\& \ y)$ `x or y, and not both`

$(x \ || \ y) \ \&\& \ (\neg x \ || \ \neg y)$ `x or y, and one of them not`

$\neg(\neg x \ \&\& \ \neg y) \ \&\& \ \neg(x \ \&\& \ y)$ `not none and not both`

$\neg(\neg x \ \&\& \ \neg y \ || \ x \ \&\& \ y)$ `not: both or none`

166

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

167

4. Defensive Programming

Constants and Assertions

168

Sources of Errors

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

169

The Compiler as Your Friend: Constants

Constants

- are variables with immutable value

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

- Usage: `const` before the definition

170

The Compiler as Your Friend: Constants

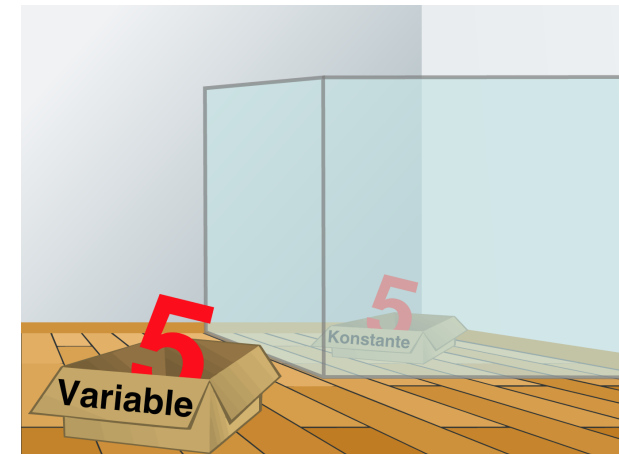
- Compiler checks that the `const`-promise is kept

```
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



171

172

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
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

173

174

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;
```

Input arguments for calculation

```
// Check validity of inputs
```

```
assert(x > 0 && y > 0); ← Precondition for the ongoing computation
```

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

175

176

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));
```

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
```

```
...
```

177

178

Fail-Fast with Assertions

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



179

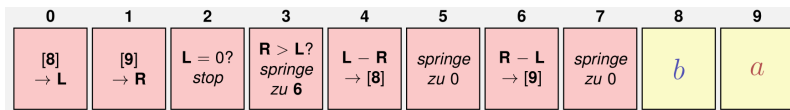
5. Control Structures I

Selection Statements, Iteration Statements, Termination, Blocks

180

Control Flow

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



Selection Statements

implement branches

- if statement
- if-else statement

181

182

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

183

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

184

Layout!

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

185

Iteration Statements

implement “loops”

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

186

Compute $1 + 2 + \dots + n$

```
// 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;
}
```

187

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

188

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)

189

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.

190

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.

191

The Solution of Gauß

- The requested number is

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

- This is half of

$$\begin{array}{r} 1 + 2 + \dots + 99 + 100 \\ + 100 + 99 + \dots + 2 + 1 \\ \hline = 101 + 101 + \dots + 101 + 101 \end{array}$$

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

192

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

193

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;
```

194

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

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

195

Example: Prime Number Test

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

A loop that can test this:

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

196

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: $n\%d \neq 0$ 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

197

Example: Correctness

```
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.

198

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";  
}
```