

# Informatics

GianAndrea Müller

May 3, 2018

## CONTENTS

|  |          |  |          |   |          |
|--|----------|--|----------|---|----------|
| <b>1 How to...</b>   | <b>2</b> | <b>5 Literals</b>  | <b>4</b> | <b>11 vector</b>                              | <b>6</b> |
| 1.1 ... <a href="#">use this summary</a> . . . . .                 | 2        | 5.1 Integer Literals . . . . .                           | 4        |   |          |
| 1.2 ... <a href="#">correct compilation errors</a> . . . . .       | 2        | 5.2 Floating-point Literals . . . . .                    | 4        | <b>12 Strings</b>                             | <b>6</b> |
| 1.3 ... <a href="#">correct runtime errors</a> . . . . .           | 2        | 5.3 Boolean Literals . . . . .                           | 4        | 12.1 C-Style Character String . . . . .       | 6        |
| 1.4 ... approach problems . . . . .                                | 2        | 5.4 Character Literals . . . . .                         | 4        | 12.2 string . . . . .                         | 6        |
| 1.5 ... find more information . . . . .                            | 2        | 5.5 String Literals . . . . .                            | 4        | <b>13 Pointers</b>                            | <b>6</b> |
| <b>2 Positional Notation</b>                                       | <b>3</b> | 5.6 Defining constants . . . . .                         | 4        | <b>14 References</b>                          | <b>6</b> |
| 2.1 Binary numbers . . . . .                                       | 3        | 5.6.1 #define . . . . .                                  | 4        | <b>15 Input/Output</b>                        | <b>6</b> |
| 2.1.1 Floating point numbers . . . . .                             | 3        | 5.6.2 const . . . . .                                    | 4        | <b>16 struct</b>                              | <b>6</b> |
| 2.2 Hexadecimal numbers . . . . .                                  | 3        | <b>6 Operators</b>                                       | <b>4</b> | <b>17 class</b>                               | <b>6</b> |
| <b>3 Syntax</b>  | <b>3</b> | 6.1 Arithmetic Operators . . . . .                       | 4        | 17.1 Class Members . . . . .                  | 6        |
| 3.1 Basic program . . . . .  | 3        | 6.2 Relational Operators . . . . .                       | 4        | 17.2 Class Access Modifiers . . . . .         | 6        |
| 3.2 Identifiers . . . . .  | 3        | 6.3 Logical Operators . . . . .                          | 4        | 17.3 Constructor and Destructor . . . . .     | 6        |
| 3.3 Comments . . . . .   | 3        | 6.4 Bitwise Operators . . . . .                          | 5        | 17.4 Copy Constructor . . . . .               | 6        |
| 3.4 Data Types . . . . .   | 3        | 6.5 Assignment Operators . . . . .                       | 5        | 17.5 friend . . . . .                         | 6        |
| 3.4.1 Primitive Types . . . . .                                    | 3        | 6.6 Misc Operators . . . . .                             | 5        | 17.6 this . . . . .                           | 6        |
| 3.4.2 Type modifiers . . . . .                                     | 4        | 6.7 Operator Precedence and As-<br>sociativity . . . . . | 5        | 17.7 Static Members . . . . .                 | 6        |
| 3.4.3 <a href="#">Find type sizes on your<br/>system</a> . . . . . | 4        | 6.7.1 How to use this table . . . . .                    | 5        | <b>18 Inheritance</b>                         | <b>6</b> |
| 3.4.4 Type qualifiers . . . . .                                    | 4        | <b>7 Loop Types</b>                                      | <b>5</b> | 18.1 Access Control and Inheritance . . . . . | 6        |
| 3.4.5 Storage classes . . . . .                                    | 4        | 7.1 while . . . . .                                      | 5        | <b>19 Overloading</b>                         | <b>6</b> |
| 3.4.6 Lvalues and Rvalues . . . . .                                | 4        | 7.2 for . . . . .  | 5        | 19.1 Function overloading . . . . .           | 6        |
| 3.4.7 Variable Definition . . . . .                                | 4        | 7.3 do...while . . . . .                                 | 5        | 19.2 Operator overloading . . . . .           | 6        |
| 3.4.8 union . . . . .  | 4        | 7.4 Loop Control Statements . . . . .                    | 5        | 19.2.1 Overloadable operators . . . . .       | 6        |
| 3.4.9 enum . . . . .   | 4        | <b>8 Conditional Statements</b>                          | <b>5</b> | <b>20 Polymorphism</b>                        | <b>6</b> |
| <b>4 Variable Scope</b>  | <b>4</b> | 8.1 if . . . . .   | 5        | <b>21 Dynamic Memory</b>                      | <b>6</b> |
| 4.1 Local Variables . . . . .                                      | 4        | 8.2 if...else . . . . .                                  | 5        | <b>22 Namespaces</b>                          | <b>6</b> |
| 4.2 Global Variables . . . . .                                     | 4        | 8.3 switch . . . . .                                     | 5        | <b>23 Templates</b>                           | <b>6</b> |
|  |          | 8.4 ? : Operator . . . . .                               | 5        | <b>24 Preprocessor</b>                        | <b>6</b> |
|  |          | <b>9 Functions</b>                                       | <b>5</b> | <b>25 Signal Handling</b>                     | <b>6</b> |
|  |          | 9.1 Structure . . . . .                                  | 5        | <b>26 Standard Template Library</b>           | <b>6</b> |
|  |          | 9.2 Declaration and Definition . . . . .                 | 5        | <b>27 Libraries</b>                           | <b>6</b> |
|  |          | 9.3 Calling a function . . . . .                         | 6        | 27.1 iostream . . . . .                       | 6        |
|  |          | 9.4 Function Arguments . . . . .                         | 6        | 27.2 math . . . . .                           | 6        |
|  |          | 9.4.1 Call by Value . . . . .                            | 6        |   |          |
|  |          | 9.4.2 Call by Reference . . . . .                        | 6        |   |          |
|  |          | 9.4.3 Default Values for Pa-<br>rameters . . . . .       | 6        |   |          |
|  |          | 9.5 Recursion . . . . .                                  | 6        |   |          |
|  |          | 9.6 Inline Functions . . . . .                           | 6        |   |          |
|  |          | <b>10 Arrays</b>   | <b>6</b> |   |          |

# 1 HOW TO...

## 1.1 ... USE THIS SUMMARY

This summary is an overview of the functionality of C++ in connection with the informatics course for mechanical engineers. It covers the content of the lectures but also contains additional information.

To emphasize the connection to the lecture all chapters containing purely additional information are marked in blue.

## 1.2 ... CORRECT COMPILATION ERRORS

Read error messages, review basic syntax, look for the additional semicolon.

## 1.3 ... CORRECT RUNTIME ERRORS

Use a [debugger](#).

## 1.4 ... APPROACH PROBLEMS

1. Define your problem.
2. Find your algorithm.
3. Code feature.
4. Compile.
5. `goto 3`.

## 1.5 ... FIND MORE INFORMATION

- [Comprehensive Tutorial](#)
- [User friendly documentation](#)
- [Extensive technical documentation](#)

# TERMS

**Algorithm** An algorithm is a set of rules that defines a sequence of operations to get to the solution of a problem.

**Language** A programming language is a set of instructions for a computer that can be used to write programs that implement algorithms.

**Syntax** The syntax of a computer language is the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language. Is a program grammatically correct?

**Semantics** The semantics of a computer language define how the language has to be interpreted. What is the meaning of a certain program?

**Editor** A program that allows writing code. There exist powerful editors that can check syntactical correctness on the fly.

**Compiler** A compiler translates a program written in a programming language to machine code, such that it can be executed by the machine.

**Computer** A computer is a device that is capable of executing machine code.

**Comments** Comments document the implemented algorithm within the program for the reader. They are ignored by the compiler.

**Include Directives** Include directives specify the additional libraries used in a program.

**The main function** The main function exists in every cpp-program. It is unique and contains the all instructions necessary to execute the program.

**Statement** Statements are the building blocks of a program. They are executed sequentially and end with a semicolon.

**Declaration** A declaration introduces a new name to the program.

**Definition** A definition introduces a body to a name within the program.

**Initialization** An initialization introduces a value to a defined name and body.

**Literals** Literals represent constant value within the program. They have a defined type and value.

**Variables** Variables represent possibly changing values within the program. They have name, type, value and address.

**Objects** Objects represent values in the computer memory. They have type, adress and value. They can be named, but can also be anonymous. Described less generally an object can be a variable, a data structure, a function, or a method.

**Expressions** Expressions represent calculations. They are a combination of values, literals, operators and functions. They are primary if they consist of a single name/literal. Otherwise they are compound. They have type and value.

**Lvalue** An lvalue is a changeable expression that has an address.

**Rvalue** An rvalue is an expression that is not an lvalue. An rvalue cannot be changed. Every lvalue can be used as an rvalue but not vice-versa.

**Operator** An operator connects expressions to compound expressions. It specifies the expected operand in type and if it is an rvalue or an lvalue. Operators have an arity.

**Arity** Arity is the number of arguments or operands an operator or a function takes. For example there exist unary and binary operators.

## 2 POSITIONAL NOTATION

### 2.1 BINARY NUMBERS

#### 2.1.1 FLOATING POINT NUMBERS

### 2.2 HEXADECIMAL NUMBERS

## 3 SYNTAX

### 3.1 BASIC PROGRAM

```
1 #include <iostream>
2 //#include "local_header_file.h"
3
4 /*
5  * Function declarations (and definitions)
6  */
7
8 int main(int argc, char ** argv)
9 {
10     /*
11      * Function calls
12      */
13     std::cout << "Hello World!" << std::endl;
14     return 0;
15 }
16
17 /*
18  * Function definitions
19  */
```

### 3.2 IDENTIFIERS

A valid identifier, i.e. the name of a variable is:

- an arbitrarily long sequence of digits, underscores, lowercase and uppercase Latin letters.
- not starting with a digit.
- not starting with two or more underscores.
- not starting with an underscore followed by a capital letter.
- not a [keyword](#) of cpp.

More information on [Identifiers](#).

### 3.3 COMMENTS

C++ allows masking code such that it is not interpreted as part of the program. This enables documenting the program. There are different possibilities:

```
1 // normal comment
2
3 /*
4 multi
5 line
6 comment
7 */
```

Both versions can be nested:

```
1 ///*comment in a comment*/
2
3 /*
4 cout<<"Hello World!<<endl; //comment in a comment
5 */
```

### 3.4 DATA TYPES

#### 3.4.1 PRIMITIVE TYPES

| Type                  | Keyword |
|-----------------------|---------|
| Boolean               | bool    |
| Character             | char    |
| Integer               | int     |
| Floating point        | float   |
| Double floating point | double  |
| Valueless             | void    |

### 3.4.2 TYPE MODIFIERS

There exist a number of type modifiers:

| Modifier | Effect                                      |
|----------|---|
| signed   | variable interpreted as signed              |
| unsigned | variable interpreted as unsigned            |
| short    | half number of allocated bits if possible   |
| long     | double number of allocated bits if possible |

Based on the primitive types and their modifiers the spectrum of available types can be established. Their sizes differ depending on compiler and environment.

| Modifier           | Typical Bit Width | Typical Range                   |
|--------------------|-------------------|---------------------------------|
| char               | 1byte             | -127 to 127                     |
| unsigned char      | 1byte             | 0 to 255                        |
| signed char        | 1byte             | -127 to 127                     |
| int                | 4byte             | -2'147'483'648 to 2'147'483'647 |
| unsigned int       | 4bytes            | 0 to 4'294'967'295              |
| signed int         | 4bytes            | -2'147'483'648 to 2'147'483'647 |
| short int          | 2bytes            | -32'768 to 32'767               |
| unsigned short int | 2bytes            | 0 to 65'535                     |
| signed short int   | 2bytes            | -32'768 to 32'767               |
| long int           | 4bytes            | -2'147'483'648 to 2'147'483'647 |
| signed long int    | 4bytes            | -2'147'483'648 to 2'147'483'647 |
| unsigned long int  | 4bytes            | 0 to 4'294'967'295              |
| float              | 4bytes            | +/- 3.4e +/- 38 ( 7 digits)     |
| double             | 8bytes            | +/- 1.7e +/- 308 ( 15 digits)   |
| long double        | 8bytes            | +/- 1.7e +/- 308 ( 15 digits)   |

### 3.4.3 FIND TYPE SIZES ON YOUR SYSTEM

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     cout<< "Size: char : "<<sizeof(char)<<endl;
6     cout<< "Size: int : "<<sizeof(int)<<endl;
7     cout<< "Size: short int : "<<sizeof(short int)<<endl;
8     cout<< "Size: long int : "<<sizeof(long int)<<endl;
9     cout<< "Size: float : "<<sizeof(float)<<endl;
10    cout<< "Size: double : "<<sizeof(double)<<endl;
11
12    return 0;
13 }
```

### 3.4.4 TYPE QUALIFIERS

### 3.4.5 STORAGE CLASSES

### 3.4.6 LVALUES AND RVALUES

### 3.4.7 VARIABLE DEFINITION

### 3.4.8 UNION

### 3.4.9 ENUM

## 4 VARIABLE SCOPE

### 4.1 LOCAL VARIABLES

### 4.2 GLOBAL VARIABLES

## 5 LITERALS

### 5.1 INTEGER LITERALS

### 5.2 FLOATING-POINT LITERALS

### 5.3 BOOLEAN LITERALS

### 5.4 CHARACTER LITERALS

### 5.5 STRING LITERALS

### 5.6 DEFINING CONSTANTS

#### 5.6.1 #DEFINE

#### 5.6.2 CONST

## 6 OPERATORS

### 6.1 ARITHMETIC OPERATORS

### 6.2 RELATIONAL OPERATORS

### 6.3 LOGICAL OPERATORS

## 6.4 BITWISE OPERATORS

## 6.5 ASSIGNMENT OPERATORS

## 6.6 MISC OPERATORS

## 6.7 OPERATOR PRECEDENCE AND ASSOCIATIVITY

| P. | Operator        | Description                            | Associativity |
|----|-----------------|--|---------------|
| 1  | ::              | Scope resolution                       | Left-to-right |
| 2  | a++ a--         | Suffix/postfix increment and decrement |               |
|    | type() type{}   | Functional cast                        |               |
|    | a()             | Function call                          |               |
|    | a[]             | Subscript                              |               |
|    | . ->            | Member access                          |               |
| 3  | ++a --a         | Prefix increment and decrement         | Right-to-left |
|    | +a -a           | Unary plus and minus                   |               |
|    | ! ~             | Locigal NOT and bitwise NOT            |               |
|    | (type)          | C-style cast                           |               |
|    | *a              | Dereference                            |               |
|    | &a              | Adress-of                              |               |
|    | sizeof          | Size-of                                |               |
|    | new new[]       | Dynamic memory allocation              |               |
|    | delete delete[] | Dynamic memory deallocation            |               |
| 4  | .* ->*          | Pointer-to-member                      | Left-to-right |
| 5  | a*b a/b a%b     | Multiplication, division, remainder    |               |
| 6  | a+b a-b         | Addition and subtraction               |               |
| 7  | << >>           | Bitwise left shift and right shift     |               |
| 8  | <=>             | Three-way comparison operator          |               |
| 9  | < <= > >=       | Relational operators                   |               |
| 10 | == !=           | Relational operators                   |               |
| 11 | &               | Bitwise AND                            |               |
| 12 | ^               | Bitwise XOR                            |               |
| 13 |                 | Bitwise OR                             |               |
| 14 | &&              | Logical AND                            |               |
| 15 |                 | Logical OR                             |               |
| 16 | a?b:c           | Ternary conditional                    | Right-to-left |
|    | throw           | throw operator                         |               |
|    | =               | Direct assignment                      |               |
|    | += -=           | Compound assignments                   |               |
|    | *= /= %=        |  |               |
|    | <<= >>=         |  |               |
|    | &= ^=  =        |  |               |
| 17 | ,               | Comma                                  | Left-to-right |

### 6.7.1 HOW TO USE THIS TABLE

```
1 cout<<a&&b;      //(cout<<a)&&b;
2
3 *p++              /*(p++);
4
5 a = b = c = d;    //a = (b =(c = d));
6
7 a + b - c;        //(a + b) - c;
8
9 delete ++*p;      //delete(++(*p))
```

1. By its precedence << is evaluated before &&.
2. By its precedence ++ is evaluated before \*.
3. Operators with the same precedence are evaluated based on their associativity. For right-to-left associative operators as =, the evaluation proceeds from right to left.  
Thus the assignments made in line 5 are in the order of their execution: `c = d;` which returns a reference to `c`, `b = c;` which returns a reference to `b` and `a = b;`.
4. Operators with the same precedence are evaluated based on their associativity. For left-to-right associative operators as + and - the evaluation proceeds from left to right.
5. `++()`, `*` and `delete` have the same precedence, and are thus evaluated based on their associativity, which is right-to-left. Therefore `++()` is evaluated after `*` and `delete` is evaluated last.

## 7 LOOP TYPES

### 7.1 WHILE

### 7.2 FOR

### 7.3 DO...WHILE

### 7.4 LOOP CONTROL STATEMENTS

## 8 CONDITIONAL STATEMENTS

### 8.1 IF

### 8.2 IF...ELSE

### 8.3 SWITCH

### 8.4 ? : OPERATOR

## 9 FUNCTIONS

### 9.1 STRUCTURE

### 9.2 DECLARATION AND DEFINITION

9.3 CALLING A FUNCTION

9.4 FUNCTION ARGUMENTS

9.4.1 CALL BY VALUE

9.4.2 CALL BY REFERENCE

9.4.3 DEFAULT VALUES FOR PARAMETERS

9.5 RECURSION

9.6 INLINE FUNCTIONS

## 10 ARRAYS

## 11 VECTOR

## 12 STRINGS

12.1 C-STYLE CHARACTER STRING

12.2 STRING

## 13 POINTERS

## 14 REFERENCES

## 15 INPUT/OUTPUT

## 16 STRUCT

## 17 CLASS

17.1 CLASS MEMBERS

17.2 CLASS ACCESS MODIFIERS

17.3 CONSTRUCTOR AND DESTRUCTOR

17.4 COPY CONSTRUCTOR

17.5 FRIEND

17.6 THIS

17.7 STATIC MEMBERS

## 18 INHERITANCE

18.1 ACCESS CONTROL AND INHERITANCE

## 19 OVERLOADING

19.1 FUNCTION OVERLOADING

19.2 OPERATOR OVERLOADING

19.2.1 OVERLOADABLE OPERATORS

## 20 POLYMORPHISM

## 21 DYNAMIC MEMORY

## 22 NAMESPACES

## 23 TEMPLATES

## 24 PREPROCESSOR

## 25 SIGNAL HANDLING

## 26 STANDARD TEMPLATE LIBRARY

## 27 LIBRARIES

27.1 IOSTREAM

27.2 MATH

27.3 CTIME