Real-Time C++

Christopher Kormanyos

Real-Time C++

Efficient Object-Oriented and Template Microcontroller Programming

Third Edition

Springer

Christopher Kormanyos
Reutlingen
Germany

ISBN 978-3-662-56717-3 ISBN 978-3-662-56718-0 (eBook)
https://doi.org/10.1007/978-3-662-56718-0

Library of Congress Control Number: 2018935981
1st edition: © Springer-Verlag Berlin Heidelberg 2013
2nd edition: © Springer-Verlag Berlin Heidelberg 2015

© Springer-Verlag GmbH Germany, part of Springer Nature 2018
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by the registered company Springer-Verlag GmbH, DE part of Springer Nature.
The registered company address is: Heidelberger Platz 3, 14197 Berlin, Germany

To those who pursue the art of technical creativity

Preface to the Third Edition

C++ is a modern, expressive object-oriented programming language that continues to evolve. In keeping up with the exciting development of C++, the third edition of this book has been updated for C++17.[1]

With this iteration of the language, the purpose of this book remains the same—to show through example and text how to leverage C++'s powerful object-oriented and template features in the realm of microcontroller programming with the goal of improving software quality and robustness while simultaneously fulfilling efficiency requirements.

Several new sections have been added and others have been modified or adapted. These changes cover new language elements and library features in C++17. They also reflect the trend of improved compiler support for C++11 and C++14.

More errors have been identified, predominantly reported by careful and patient readers. All errors that have been found have been corrected.

New or Significantly Modified Sections

The third edition of this book contains several new or significantly modified sections. These include:

- Section 2.2 updated for a newer GCC toolchain with a more simple decorated name (i.e., GCC version 7.2.0 built for the target `avr-g++`),
- Section 3.4 adding information on C++17 nested namespace definitions,
- Section 3.17 now including descriptions of the (in the second edition of this book missing) standardized suffixes `if`, `i` and `il` from the `<complex>` library,
- Section 3.18 (new) detailing the specifiers **`alignof`** and **`alignas`**,

[1]At the time of writing the third edition of this book, state-of-the-art compilers support C++17. The specification process is ongoing, and some language experts predict that C++20 will be the next revision of the C++ standard, potentially available in 2020.

- Section 3.19 (new) for the specifier **`final`**,
- Section 3.20 (new) on defining types with C++11 alias,
- Section 9.8 (new) portraying a full example that animates an RGB LED to produce a colorful light display,
- Section 12.4 covering inclusion of additional mathematical special functions in `<cmath>` specified in the C++17 standard,
- several sections in Chap. 13 reflecting improvements of the `fixed_point` class in the companion code,
- Section 16.6 (new) presenting an extended-complex template class that promotes the functionality of the `<complex>` library to user-defined types other than **`float`**, **`double`** and **`long double`**,
- Chapter 17 (new) showing how to use C code in a C++ project (hereby "Additional Reading" has been moved from Chaps. 17 to 18),
- the tutorial of Appendix A, in particular Sect. A.4 updating `static_assert` for C++17, Sect. A.15 (new) for the `<type_traits>` library, Sect. A.16 (new) on using `std::any` from the C++17 `<any>` library, and Sect. A.17 (new) introducing structured binding declarations (also from C++17).

Improved or New Examples and Code Snippets

All sample projects have been modernized for GCC version 7.2.0 built for `avr-g++` and five new examples have been added.

☞ The `chapter06_01` sample project (new) shows step-by-step how to perform the benchmark of the CRC calculation described in Sects. 6.1 and 6.2.

☞ The `chapter09_07` example in Sect. 9.7 has been adapted to architectural improvements found in the new `chapter09_08` sample of Sect. 9.8,

☞ The `chapter09_08` sample project (new) animates an industry-standard off-the-shelf RGB LED. This example incorporates several real-time C++ features including object-oriented design, peripheral driver development and multitasking. They are merged together within the context of a coherent, intuitive and visible project. By means of simulation on a PC, the `chapter09_08` sample also exemplifies cross-development and methods for creating portable code.

☞ The `chapter12_04` example (new) performs highly detailed calculations of several mathematical special functions. These are used to provide a benchmark of floating-point operations.

☞ The `chapter17_03` sample project (new) takes an existing C library used for CRC calculations and wraps the procedural functions in classes that can be employed in object-oriented C++. This practical exercise shows how to leverage the power of valuable existing C code within a modern C++ project.

☞ The `chapter17_03a` sample project (new) uses the CRC classes of the `chapter17_03` example and distributes the work of the calculations among successive time slices in a multitasking environment.

With the third edition of this book, code snippets have been made available in the public domain. The code snippets correspond to certain code samples that appear in the text. Each code snippet comprises a complete and portable, single-file C++ program. Every program can be compiled and run on a PC or easily adapted to a microcontroller environment.

To obtain run-ability on a PC, code snippets have been embellished with a `main()` subroutine. Some code snippets have been augmented with `<thread>` support or other higher-level mechanisms in order to elucidate the topic of the program. Outputs are printed to the console with `<iostream>`. The file names of the code snippets correspond to chapter and section numbers in the book.

Companion Code

The companion code has been improved and extended based on new and reworked sections of the third edition. Contemporary compiler toolchains are used. Legacy directories that previously provided for certain aspects of C++11 compatibility have been removed, as modern compilers now support these.

The entire companion code can be found here:

http://github.com/ckormanyos/real-time-cpp

The reference application is at:

http://github.com/ckormanyos/real-time-cpp/tree/master/ref_app

Example projects can be found here:

http://github.com/ckormanyos/real-time-cpp/tree/master/examples

Code snippets are located at:

http://github.com/ckormanyos/real-time-cpp/tree/master/code_snippets

Further Notes on Coding Style

The coding style in the third edition of this book stays consistent with that used in the first and second editions. The code is intended to be easy to read and straightforward to comprehend while simultaneously utilizing the full spectrum of C++'s traditional and modern features.

Updated Trademarks and Acknowledgments

In the preface to first edition of this book, we listed several trademarks and acknowledgments. Meanwhile the authors/holders of certain trademarks/copyrights and the scope of some of the acknowledgments have changed.

- ATMEL® and AVR® are registered trademarks of Microchip Technology Incorporated or its subsidiaries in the US and other countries.
- *Real-Time C++*: *Efficient Object-Oriented and Template Microcontroller Programming* is a book authored by Christopher Kormanyos and published by Springer Verlag and has not been authorized, sponsored, or otherwise approved of by Microchip Technology Incorporated.
- ARDUINO® is a registered trademark of the Arduino Group.
- The word AUTOSAR is a registered trademark of the AUTOSAR Development Partnership.
- SPI™ is a trademark of Motorola Corporation.
- The circuits of all target hardware described in this book and depicted in various chapters such as Chaps. 2, 9 and Appendix D were designed and assembled on solderless prototyping breadboards by Christopher Kormanyos.
- All photographs of target hardware in this book shown in chapters including Chaps. 2, 9, Appendix D and any others were taken by Christopher Kormanyos.

Reutlingen, Germany Christopher Kormanyos
February 2018

Preface to the Second Edition

C++ seamlessly blends object-oriented techniques with generic template methods, creating a modern powerful programming language useful for problem-solving in countless domains. The most recent evolution of C++ from C++11 to C++14 has brought yet further improvements to this rich language.[2] As C++ becomes even more expressive, growing numbers of embedded systems developers are discovering new and fascinating ways to utilize its multifaceted capabilities for creating efficient and effective microcontroller software.

The second edition of this book retains its original purpose to serve as a practical guide to programming real-time embedded microcontroller systems in C++. New material has been incorporated predominantly reflecting changes introduced in the C++14 standard. Various sections have been reworked according to reader suggestions. Selected passages have been reformulated in a continued effort to improve clarity. In addition, all known errors throughout the text have been corrected.

New sections have been added (in particular for C++14) covering:

- digit separators (Sect. 3.15),
- binary literals (Sect. 3.16),
- user-defined literals (Sect. 3.17),
- variable templates (Sect. 5.12),
- and the `chapter09_07` sample project (Sect. 9.7) controlling an industry-standard seven-segment display.

Two new sample projects, `chapter02_03a` and `chapter09_07`, have been added to the companion code.

[2]At the time of writing the second edition of this book, C++14 is brand new. World-class compilers are shipped with support for C++14. Work is in progress on C++1z, the next specification of C++ (sometimes known as C++17). Experts anticipate that the specification of C++1z could be finished in 2017.

☞ The `chapter02_03a` sample project implements LED toggling at 1/2 Hz with timing provided by a simple multitasking scheduler in combination with a timer utility.
☞ The `chapter09_07` sample project in the newly added Sect. 9.7 uses many of the advanced programming methods in this book to animate an industry-standard seven-segment display.

Significantly reworked or corrected parts of this book include:

- corrections and clarifications in Chap. 1 on getting started with C++,
- the description of the `chapter02_02` project in Sect. 2.2,
- parts of Chap. 3 on the jump-start in real-time C++,
- corrections and clarifications in Chap. 5 on templates,
- Sections 6.1 and 6.2 on optimization and performance,
- parts of Chap. 10 on custom memory management,
- parts of Chaps. 12 and 13 on mathematics,
- the literature list in Sect. 18.1,
- parts of Appendix A in the C++ tutorial,
- and repairs and extensions of the citations in some chapter references.

Companion Code

The companion code continues to be supported and numerous developers have successfully worked with it on various cross-development platforms. The scope of the companion code has been expanded to include a much wider range of target microcontrollers. In addition, the `chapter02_03a` and `chapter09_07` sample projects that are mentioned above have been added to the companion code.

The companion code is available at:
http://github.com/ckormanyos/real-time-cpp

More Notes on Coding Style

The second edition of this book features slight changes in coding style. These can be encountered in the code samples throughout the text.

Compiler support for standard C99 and C++11 macros of the form `UINT8_C()`, `UINT16_C()`, `UINT32_C()`, etc. and corresponding macros for signed types in the `<stdint.h>` and `<cstdint>` headers has become more prevalent (see also Sect. 3.2). Consequently, these macros are used more frequently throughout the code samples.

These macros are useful for creating integer numeric literal values having specified widths. The code below, for example, utilizes `UINT8_C()` to initialize an 8-bit integer variable with a numeric literal value.

```
#include <cstdint>

std::uint8_t byte_value = UINT8_C(0x55);
```

Digit separators have become available with C++14 (Sect. 3.15). These are used in selected code samples to improve clarity of long numeric literals. Digit separators are shown in the code sample below.

```
#include <cstdint>

constexpr std::uint32_t prime_number =
  UINT32_C(10'006'721);

constexpr float pi = 3.1415926535'8979323846F;
```

Other than these minor changes, however, the coding style in the second edition of this book remains consistent with that of the first edition and is intended to be clean and clear.

Reutlingen, Germany
Seattle, Washington
May 2015

Christopher Kormanyos

Preface to the First Edition

This book is a practical guide to programming real-time embedded microcontroller systems in C++. The C++ language has powerful object-oriented and template features that can improve software design and portability while simultaneously reducing code complexity and the risk of error. At the same time, C++ compiles highly efficient native code. This unique and effective combination makes C++ well suited for programming microcontroller systems that require compact size, high performance and safety-critical reliability.

The target audience of this book includes hobbyists, students and professionals interested in real-time C++. The reader should be familiar with C or another programming language and should ideally have had some exposure to microcontroller electronics and the performance and size issues prevalent in embedded systems programming.

About This Book

This is an interdisciplinary book that includes a broad range of topics. Real-world examples have been combined with brief descriptions in an effort to provide an intuitive and straightforward methodology for microcontroller programming in C++. Efficiency is always in focus and numerous examples are backed up with real-time performance measurements and size analyses that quantify the true costs of the code down to the very last byte and microsecond.

Throughout the chapters, C++ is used in a bare-bones, no-frills fashion without relying on any libraries other than those specified in the language standard itself. This approach facilitates portability.

This book has three parts and several appendices. The three parts generally build on each other with the combined goal of providing a coherent and effective set of C++ methods that can be used with a wide range of embedded microcontrollers.

- Part I provides a foundation for real-time C++ by covering language technologies. Topics include getting started in real-time C++, object-oriented methods, template programming and optimization. The first three chapters have a particularly hands-on nature and are intended to boost competence in real-time C++. Chapter 6 has a unique and important role in that it is wholly dedicated to optimization techniques appropriate for microcontroller programming in C++.
- Part II presents detailed descriptions of a variety of C++ components that are widely used in microcontroller programming. These components can be either used as presented or adapted for other projects. This part of the book uses some of C++'s most powerful language elements, such as class types, templates and the STL, to develop components for microcontroller register access, low-level drivers, custom memory management, embedded containers, multitasking, etc.
- Part III describes mathematical methods and generic utilities that can be employed to solve recurring problems in real-time C++.
- The appendices include a C++ language tutorial, information on the real-time C++ development environment and instructions for building GNU GCC cross-compilers and a microcontroller circuit.

C++ is a rich language with many features and details, the description of which can fill entire bookshelves. This book, however, primarily concentrates on how to use C++ in a real-time microcontroller environment. Along those lines, C++ language tutorials have been held terse, and information on microcontroller hardware and compilers is included only insofar as it is needed for the examples. A suggested list of additional reading material is given in Chap. 18 for those seeking supplementary information on C++, the C++ standard library and STL, software design, C++ coding guidelines, the embedded systems toolchain and microcontroller hardware.

When units are needed to express physical quantities, the MKS (meter, kilogram, second) system of units is used.

Companion Code, Targets and Tools

The companion code includes three introductory projects and one reference project. The introductory projects treat various aspects of the material presented in Chaps. 1 and 2. The reference project is larger in scope and exercises many of the methods from all the chapters.

The companion code is available at:

http://github.com/ckormanyos/real-time-cpp

The C++ techniques in this book specifically target microcontrollers in the *small-to-medium* size range. Here, small-to-medium spans the following approximate size and performance ranges.

- 4 kbyte ... 1 Mbyte program code
- 256 byte ... 128 kbyte RAM

- 8-bit . . . 32-bit CPU
- 8 MHz . . . 200 MHz CPU frequency

Most of the methods described in this book are, however, scalable. As such, they can be used equally well on larger or smaller devices, even on PCs and workstations. In particular, they can be employed if the application has strict performance and size constraints.

A popular 8-bit microcontroller clocked with a frequency of 16 MHz has been used as the primary target for benchmarking and testing the code samples in this book. Certain benchmarks have also been performed with a well-known 32-bit microcontroller clocked at 24 MHz. An 8-bit microcontroller and a 32-bit microcontroller have been selected in order to exercise the C++ methods over a wide range of microcontroller performance.

All the C++ examples and benchmarks in the book and the companion code have been compiled with GNU GCC versions 4.6.2 and 4.7.0. Certain examples and benchmarks have also been compiled with other PC-based compilers.

The most recent specification of C++11 in ISO/IEC 14882:2011 is used throughout the text. At the time this book is written, the specification of C++11 is brand new. The advent of C++11 has made C++ significantly more effective and easy to use. This will profoundly influence C++ programming. The well-informed reader will, therefore, want to keep in touch with C++11 best practice as it evolves in the development community.

Notes on Coding Style

A consistent coding style is used throughout the examples in this book and in the companion code.

Code samples are written with a `fixed-width font`. C++ language keywords and built-in types use the same font, but they are in boldface. For instance,

```
constexpr int version = 7;
```

In general, the names of all symbols such as variables, class types, members and subroutines are written in lowercase. A single underscore (_) is used to separate words and abbreviations in names. For instance, a system-tick variable expressed with this style is shown in the code sample below.

```
unsigned long system_tick;
```

Using prefixes, suffixes or abbreviations to incorporate type information in a name, sometimes known as *Hungarian notation*, is not done. Superfluous prefixes, suffixes and abbreviations in Hungarian notation may obscure the name of a symbol and symbol names can be more intuitive and clear without them. For example,

```
std::uint16_t name_of_a_symbol;
```

Names that are intended for use in public domains are preferentially long and descriptive rather than short and abbreviated. Here, clarity of expression is preferred over terseness. Symbols used for local subroutine parameters or private implementation details with obvious meanings, however, often have terse or abbreviated names.

The global subroutine below, for example, uses this naming style. It returns the **float** value of the squared Euclidean distance from the origin of a point in two-dimensional Cartesian space $\mathbb{R}^2$.

```
float squared_euclidean_distance(const float& x,
                                 const float& y)
{
  return (x * x) + (y * y);
}
```

C++ references are heavily used because this can be advantageous for small microcontrollers. Consider an 8-bit microcontroller. The work of copying subroutine parameters or the work of pushing them onto the stack for anything wider than 8 bits can be significant. This work load can potentially be reduced by using references. In the previous code sample, for instance, the floating-point subroutine parameters x and y, each four bytes wide, have been passed to the subroutine by reference (i.e., **const float**&).

Fixed-size integer types defined in the std namespace of the C++ standard library such as std::uint8_t, std::uint16_t, std::uint32_t, and the like are preferentially used instead of plain built-in types such as **char**, **short**, **int**, etc. This improves clarity and portability. An unsigned login response with exactly 8 bits, for instance, is shown below.

```
std::uint8_t login_response;
```

Code samples often rely on one or more of the C++ standard library headers such as <algorithm>, <array>, <cstdint>, <limits>, <tuple>, <vector>, etc. In general, code samples requiring library headers do not explicitly include their necessary library headers.

The declaration of `login_response` above, for example, actually requires `<cstdint>` for the definition of `std::uint8_t`. The library file is, however, not included. In general, the code samples focus on the core of the code, not on the inclusion of library headers.

It is easy to guess or remember, for example, that `std::array` can be found in `<array>` and that `std::vector` is located `<vector>`. It can, however, be more difficult to guess or remember that `std::size_t` is in `<cstddef>` or that `std::accumulate()` is in `<numeric>`. With assistance from online help and other resources and with a little practice, though, it becomes routine to identify what standard library parts can be found in which headers.

In cases for which particular emphasis is placed on the inclusion of a header file, the relevant **`#include`** line(s) may be explicitly written. For instance,

```
#include <cstdint>

std::uint8_t login_response;
```

Namespaces are used frequently. In general, though, the **`using`** directive is not used to inject symbols in namespaces into the global namespace. This means that the entire namespace must be typed with the name of a symbol in it. This, again, favors non-ambiguity over brevity.

The unsigned 16-bit counter below, for example, uses a type from the `std` namespace. Since the "**`using namespace`** `std`" directive is not used, the name of the namespace (`std`) is explicitly included in the type.

```
std::uint16_t counter;
```

Suffixes are generally appended to literal constant values. When a suffix is appended to a literal constant value, its optional case is uppercase. For example,

```
constexpr float pi = 3.14159265358979323846F;

constexpr std::uint8_t login_key = 0x55U;
```

Certain established C++ coding guidelines have strongly influenced the coding style. For the sake of terseness and clarity, however, not every guideline has been followed all the time.

One clearly recognizable influence of the coding guidelines is the diligent use of C++-style casts when converting built-in types. The following code, for instance, explicitly casts from **float** to an unsigned integer type.

```
float f = 3.1415926535897932384 6F;

std::uint8_t u = static_cast<std::uint8_t>(f);
```

Even though explicit casts like these are not always mandatory, they can resolve ambiguity and eliminate potential misinterpretation caused by integer promotion.

Another influence of the coding guidelines on the code is the ordering of class members according to their access level in the class. The `communication` class below, for example, represents the base class in a hierarchy of communication objects. The members in the class definition are ordered according to access level. In particular,

```
class communication
{
public:
  virtual ~communication();

  virtual bool send(const std::uint8_t) const;
  virtual bool recv(std::uint8_t&);

protected:
  communication();

private:
  bool recv_ready;
  std::uint8_t recv_buffer;
};
```

C-style preprocessor macros are used occasionally. Preprocessor macros are written entirely in uppercase letters. Underscores separate the words in the names of preprocessor macros. The `MAKE_WORD()` preprocessor macro below, for example, creates an unsigned 16-bit word from two unsigned 8-bit constituents.

```
#define MAKE_WORD(lo, hi) \
  (uint16_t) (((uint16_t) (hi) << 8) | (lo))
```

Acknowledgements

First and foremost, I would like to thank my wife and my daughter for encouraging me to write this book and also for creating a peaceful, caring atmosphere in which I could work productively. Thank you for your support and your time. You have my gratitude.

I would also like to express appreciation to family, friends and associates, too numerous to list, who contributed to this project with their innovative ideas, support, friendship and companionship.

Thanks go to the members of the C++ standards committee, Boost, the volunteers at GCC and all the developers in the vibrant C++ and embedded systems communities. Through your efforts, oftentimes for no pay whatsoever, C++ has evolved to an unprecedented level of expressiveness, making object-oriented and generic programming more effective and easier than ever.

Working with Springer Verlag was a delightful experience. I thank my editor, who first identified the merit of this work and supported me throughout the writing process. I also thank the copy editing team and all the staff at Springer Verlag for their professionalism and capable assistance.

- ATMEL® and AVR® are registered trademarks of Atmel Corporation or its subsidiaries, in the US and other countries.
- *Real-Time C++: Efficient Object-Oriented and Template Microcontroller Programming* is a book authored by Christopher Kormanyos and published by Springer Verlag and has not been authorized, sponsored, or otherwise approved of by Atmel Corporation.
- ARDUINO® is a registered trademark of the Arduino Group.
- SPI™ is a trademark of Motorola Corporation.
- The circuit of the target hardware described in this book and depicted in Chap. 2 and Appendix D was designed and assembled on a solderless prototyping breadboard by Christopher Kormanyos.
- The photographs of the target hardware described in this book and depicted in Chap. 2 and Appendix D were taken by Christopher Kormanyos.

Reutlingen, Germany Christopher Kormanyos
Seattle, Washington
September 2012

Contents

Acronyms

$\mathbb{C}$	$\mathbb{C}$ represents the set of complex numbers in mathematics.
$\mathbb{R}$	$\mathbb{R}$ represents the set of real numbers on the real axis in mathematics.
$\mathbb{R}^2$	$\mathbb{R}^2$ represents two-dimensional Cartesian space in mathematics and geometry.
$\mathbb{R}^3$	$\mathbb{R}^3$ represents three-dimensional Cartesian space in mathematics and geometry.
$\mathbb{Z}$	$\mathbb{Z}$ represents the set of integer numbers in mathematics.
ADC	Analog-Digital Converter.
ASCII	American Standard Code for Information Interchange [25] is a numerical representation of characters, often used in areas such as computer programming and telecommunication.
AUTOSAR	AUTomotive Open System ARchitecture [2] is a worldwide cooperation of automotive manufacturers and companies supplying electronics, semiconductors and software that concentrates on, among other things, a standardized architecture for automotive microcontroller software.
AWG	American Wire Gauge.
binutils	Binary Utilities [6] are the GNU binary utilities such as archiver, assembler, linker, object file parsers, etc. for GCC.
C	C is the C programming language, which is often referred to as ANSI-C or C89 [1]. Later versions of C include C99 [13] and C11 [17].
C99	C99 refers to the C programming language, as specified in ISO/IEC 9899:1999 [13].
C11	C11 refers to the C programming language, as specified in ISO/IEC 9899:2011 [17].
C++	C++ refers to the C++ programming language.
C++98	C++98 refers to the C++ programming language, as specified in ISO/IEC 14882:1998 [12].
C++03	C++03 refers to the C++ programming language, as specified in ISO/IEC 14882:2003 [15].

C++11	C++11 refers to the C++ programming language, as specified in ISO/IEC 14882:2011 [18].
C++14	C++14 refers to the C++ programming language, as specified in ISO/IEC 14882:2014 [19].
C++17	C++17 refers to the C++ programming language, as specified in ISO/IEC 14882:2017 [20].
C++20	C++20 [26] is predicted by some C++ language experts to be the next revision of the C++ standard, possibly to become available in the year 2020.
CLooG	Chunky Loop Generator [4] is a software library used for geometric polyhedron analysis.
CRC	Cyclic Redundancy Check [27].
CPU	Central Processing Unit.
ctor	constructor of a class object in object-oriented programming is a special subroutine that is called when an object is created.
DIL	Dual In-Line electronic component packaging.
DSP	Digital Signal Processor.
dtor	destructor of a class object in object-oriented programming is a special subroutine that is called when an object is destroyed or deleted.
FIR	Finite-Impulse Response is a kind of digital filter.
FLASH	Flash Memory is a nonvolatile computer memory that can be electrically written and erased. Flash is commonly used as an alternative to ROM.
FPU	Floating-Point Unit implements floating-point arithmetic in hardware. Many modern high-performance microcontrollers use an FPU to accelerate floating-point calculations.
GAS	is the GNU ASsembler.
GCC	GNU Compiler Collection [7] is a collection of free compilers for several popular programming languages including, among others, C and C++. GCC is supported for a wide range of targets.
GMP	GMP is the GNU Multiple-Precision library [9]. It implements highly efficient multiple-precision representations of integer and floating-point data types.
GNU	Is a $*$nix-like computer operating system consisting entirely of free software [8].
GUI	Graphical User Interface.
HEX	Hexadecimal representation is a base 16 numerical representation commonly used to store program data in computer engineering.
ICE	In-Circuit Emulator is a highly sophisticated hardware device used to debug embedded microcontroller software with an emulated bond-out processor.
ISL	Integer Set Library [11] is a software library used for manipulating sets of integers.

ISP	In-System-Programming is the act of programming the program code of a microcontroller using a communication interface while the microcontroller is fitted in the application, rather than as a standalone non-soldered component.
ISR	Interrupt Service Routine.
JTAG	Joint Test Action Group, later standardized as IEEE 1149.1 [10], is a protocol and hardware interface used for printed circuit board testing, boundary scan and recently more and more for debugging embedded systems.
LED	Light-Emitting Diode is a semiconductor-based light source used in diverse applications such as lighting, consumer electronics and toys.
MCAL	Microcontroller Abstraction Layer is a low-level layer in a layered software architecture (such as AUTOSAR). The interface of the MCAL is typically written in a portable fashion. The MCAL implementation itself, however, contains partially non-portable components that access microcontroller peripherals and their registers, such as PWM signal generators, timers, serial UARTs and other communication interfaces, etc.
MinGW	Minimalist GNU [21] is an open-source programming toolset that emulates ∗nix-like environments.
MKS	Meter, Kilogram, Second is a system of units used to express physical quantities.
MPC	Multiple-Precision Complex [22] is a GNU C library that implements multiple-precision arithmetic of complex numbers.
MPFR	Multiple-Precision Floating-Point with correct Rounding [5, 23] is the GNU multiple-precision floating-point library. It is built on top of GMP and places special emphasis on efficiency and correct rounding.
MSYS	Minimal SYStem [21] is a collection of GNU utilities that enhance and extend the MinGW shell.
newlib	newlib [24] is a free implementation of the C standard library. It is well suited for use with embedded systems and has been ported to a variety of CPU architectures.
nop	No OPeration is a common assembly instruction that simply does *no operation*. One or more nops are often chained sequentially in order to be used for ultra low-level functions such as creating very short delays or flushing an instruction pipeline.
opcode	OPeration CODE is a machine language instruction containing the operation to be done.
PC	Personal Computer.
POSIX	Portable Operating System Interface is an open standardized operating system specified in ISO/IEC 9945:2003 [14].
PPL	Parma Polyhedra Library [3] is a software library for abstract geometrical polyhedron representations.
PWM	Pulse-Width Modulated signal is a square wave that usually has a fixed period and a variable duty cycle.

RAM	Random Access Memory is computer memory with nearly constant access time regardless of address or memory size. RAM is volatile in the sense that data are typically lost when the power is switched off.
ROM	Read-Only Memory is a class of computer memory that, once written, can only be modified with external programming tools—or not be modified at all. ROM has permanent character in the sense that data are retained throughout power on/off cycles.
SPI^{TM}	Serial Peripheral Interface bus is a four-wire serial communication interface commonly used for communication between a microcontroller and one or more off-chip devices on the printed circuit board.
STL	Standard Template Library is part of the C++ standard library. The standard template library contains a vast collection of generic containers, iterators and algorithms.
TO-220	Transistor Outline electronic component packaging, number 220.
TR1	C++ Technical Report 1 includes the standard library extensions that are specified in ISO/IEC TR 19768:2007 [16]. TR1 has been predominantly integrated in C++11 (ISO/IEC 14882:2011 [18]).
UART	Universal Asynchronous Receiver/Transmitter is an asynchronous receiver and transmitter commonly used for serial communication between a PC and a microcontroller.

References

1. ANSI, *ANSI X3.159-1989 American National Standard for Information Systems – Programming Language C* (American National Standard for Information, New York, 1989)
2. AUTOSAR, *Automotive Open System Architecture* (2017), http://www.autosar.org
3. BUGSENG, *Parma Polyhedra Library (PPL)* (2012), http://www.bugseng.com/products/ppl
4. CLooG, *Chunky Loop Generator* (2015), http://www.cloog.org
5. L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, P. Zimmermann, MPFR: a multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. **33**(2) (2007). Article 13
6. Free Software Foundation, *GNU Binutils* (2011), http://www.gnu.org/software/binutils
7. Free Software Foundation, *GNU Compiler Collection* (2015), http://gcc.gnu.org
8. Free Software Foundation, *GNU Operating System* (2015), http://gnu.org
9. GMP, *GNU Multiple Precision Arithmetic Library* (2012), http://gmplib.org
10. IEEE Computer Society, *IEEE Std 1149.1 – 1990: IEEE Standard Test Access Port and Boundary-Scan Architecture* (1990). Available at http://standards.ieee.org/findstds/standard/1149.1-1990.html
11. ISL, *Integer Set Library* (2015), http://isl.gforge.inria.fr
12. ISO/IEC, *ISO/IEC 14882:1998: Programming Languages – C++* (International Organization for Standardization, Geneva, 1998)
13. ISO/IEC, *ISO/IEC 9899:1999: Programming Languages – C* (International Organization for Standardization, Geneva, 1999)
14. ISO/IEC, *ISO/IEC 9945:2003: Information Technology – Portable Operating System Interface (POSIX)* (International Organization for Standardization, Geneva, 2003)
15. ISO/IEC, *ISO/IEC 14882:2003: Programming Languages – C++* (International Organization for Standardization, Geneva, 2003)

16. ISO/IEC, *ISO/IEC TR 19768:2007: Information Technology – Programming Languages – Technical Report on C++ Library Extensions* (International Organization for Standardization, Geneva, 2007)
17. ISO/IEC, *ISO/IEC 9899:2011: Programming Languages – C* (International Organization for Standardization, Geneva, 2011)
18. ISO/IEC, *ISO/IEC 14882:2011: Information Technology – Programming Languages – C++* (International Organization for Standardization, Geneva, 2011)
19. ISO/IEC, *ISO/IEC 14882:2014: Information Technology – Programming Languages – C++* (International Organization for Standardization, Geneva, 2014)
20. ISO/IEC, *ISO/IEC 14882:2017: Information Technology – Programming Languages – C++* (International Organization for Standardization, Geneva, 2017)
21. MinGW, *Home of the MinGW and MSYS Projects* (2012), http://www.mingw.org
22. MPC, *GNU MPC* (2012), http://www.multiprecision.org
23. MPFR, *GNU MPFR Library* (2013), http://www.mpfr.org
24. Red Hat, *newlib* (2013), http://sourceware.org/newlib
25. Wikipedia, *ASCII* (2017), http://en.wikipedia.org/wiki/ASCII
26. Wikipedia, *C++20* (2017), http://en.wikipedia.org/wiki/C%2B%2B20
27. Wikipedia, *Cyclic Redundancy Check* (2017), http://en.wikipedia.org/wiki/Cyclic_redundancy_check