



Improve your productivity,
reduce your debugging time.

PRAGMATIC C++ ARDUINO PROGRAMMING

Self-published via Amazon by the author - Michèle Delsol

Pragmatic C++ Arduino Programming by Michèle Delsol

Proper names, trademarks, and designations used by the author are capitalized to distinguish them from ordinary text. They are the property of their respective owners. The author and publisher of this book have no intent at establishing any relationship whatsoever with the owners of these names, trademarks, and designations.

The author and publisher have exercised due diligence as to the exactitude of the book's content and issue no explicit or implied warranty of any kind as to the suitability of content presented for any purposes whatsoever and assume no responsibility for errors or omissions. The author and publisher assume no liability for incidental or consequential damages resulting from the use of information, code snippets, and programs presented in this book.

First published via Amazon August 2023.

Delsol, Michèle

Pragmatic C++ Arduino Programming / Michèle Delsol, first edition

Copyright © 2023 by Michèle Delsol

All rights reserved. Printed via Amazon "Print-on-demand" in the United States and in other countries where Amazon distributes this book. This book is protected by United States and international copyright laws. No parts of this book may be copied in any form whatsoever; permission must be obtained to reproduce parts and the entirety of this book by any means whatsoever: electronic, photocopying, mechanical, or other.

Indexing was undertaken via JavaScript scripts applied on manually created tags. The indexing system was created by the author.

The marmoset monkey illustration on the front cover is an original pencil and China ink drawing by the author.

Paperback edition : ISBN 978-2-9585628-0-9

First edition published via Amazon August 2023 in paperback, hard cover, and electronic (Kindle) formats.

Table of contents

List of tables and figures	xv
Acknowledgements	xvii
Preface	xix
Introduction	1
Chapter 1 Arduino and C++	5
1.1 A short history of C and C++	7
1.2 C++ programming	9
1.3 Microcontrollers	10
1.4 Programming languages	12
1.5 Which chip/language combination?	14
1.6 What defines C++	16
Chapter 2 Arduino IDE	19
2.1 Arduino C++ editor	21
2.2 Preprocessor	23
2.3 Compiler	25
2.4 Linker	26
2.5 Make utility	26
2.6 Uploader (avrdude)	27
2.7 bootloader	27
2.8 Serial terminal	29
2.9 Hardware-based debugging	29
Chapter 3 Other IDEs	31
3.1 AtmelStudio	32
3.2 Visual Micro for MicrochipStudio (AtmelStudio)	32
3.3 Visual Micro for Microsoft Visual Studio	33
3.4 VS Code (Visual Studio Code)	33
3.5 PlatformIO	33
3.6 Code::Blocks	34
3.7 MPLAB	34
3.8 Visual development	35
3.8.1 Visualino	35
3.8.2 Scratch for Arduino	35
3.8.3 Blynk for Arduino	35
3.9 Artificial intelligence (ChatGPT)	35
Chapter 4 What one needs to master	37
4.1 C++ enhancements to C	38

4.2	What is a C++ program	39
4.3	What does a C++ program look like	40
4.4	What to do with setup and loop	44
4.5	Syntax differences between C and C++	45
4.6	Good programming practices	46
Chapter 5	C++ building-blocks	49
5.1	Comments	51
5.2	Constants	52
5.3	Types	53
5.3.1	Built-in types	54
5.3.2	User-defined types	56
5.4	Type qualifiers	56
5.4.1	typedef	57
5.4.2	auto	58
5.4.3	static	58
5.4.4	const and mutable	61
5.4.5	register and volatile	63
5.4.6	size_t	64
5.5	Operators	64
5.5.1	Operator precedence and associativity	67
5.5.2	Special symbol colon ':'	69
5.5.3	Special symbol double-colon '::' - scope resolution operator	69
5.5.4	sizeof operator	70
5.5.5	Bit-level operators	70
5.6	Code-blocks	72
5.7	Statements	72
5.8	Control flow statements	73
5.8.1	if (condition) {...} else {...}	74
5.8.2	for (iterate) {...}	74
5.8.3	for (each) {...}	75
5.8.4	while (condition) {...}	75
5.8.5	do {...} while (condition)	76
5.8.6	switch (value) {case1, case2, ... }	76
5.8.7	goto label	76
5.8.8	Exception handling (C) setjmp/longjmp	77
5.9	Functions and variables	78
5.9.1	Variables	81
5.10	System variables	83
5.11	Arduino specific functions	83
5.11.1	Digital and analog I/O functions	84
5.11.2	Signal functions	85
5.11.3	Timers	85

5.11.4 Interrupts	86
5.11.5 Serial communications	87
5.11.6 Random numbers	88
5.11.7 Arduino bit functions	89
5.12 class and struct	89
5.12.1 Constructors and destructors	92
5.12.2 Bitfields	93
5.13 Arrays and indices	94
5.14 Unions	96
5.15 Enumerations (enum)	97
5.16 Save RAM with PROGMEM	98
5.17 Libraries	98
Chapter 6 C++ mechanics	101
6.1 Header files (.h) and code files (.cpp, .ino)	103
6.2 Scope (visibility)	104
6.3 Increment/decrement prefix/postfix (++/--)	106
6.4 Function creation	109
6.5 Parameter passing	112
6.5.1 Pass by value	112
6.5.2 Pass by address	114
6.5.3 Pass by reference	115
6.5.4 Pass by reference saves time, money, and RAM	117
6.5.5 Pass by reference summary	118
6.6 Polymorphism	120
6.7 Compatible numeric types	121
6.8 Strings	122
6.9 class/struct array initializations	123
6.10 Arithmetic on array items	124
6.11 Templates	125
Chapter 7 What one needs to be aware of	129
7.1 Operator overloading	130
7.2 Data packing (bit-level work)	131
7.3 Inheritance	134
7.3.1 Virtual functions, pure virtual functions, abstract classes	136
7.3.2 Cycling through derived classes	137
7.4 this	139
7.5 Function pointers	140
7.6 Inlining	142
7.7 Lambda functions	143
7.8 namespace	144
7.9 Error handling	145

7.10 C++ exception handling	147
7.11 Complex numbers	150
7.12 C++ features not supported by Arduino	150
Chapter 8 Memory management	153
8.1 Arduino memory pools	154
8.2 RAM partitioning	155
8.3 How RAM use evolves	157
8.4 Managing RAM	160
8.4.1 Available RAM	161
8.4.2 Heap	164
8.4.3 Stack frames	167
8.5 Use EEPROM to store data from run to run	168
Chapter 9 Macros	169
9.1 Macro uses	169
9.2 How to create macros	171
9.3 Multiline macros	172
9.4 Macro types	172
9.4.1 #define macros	174
9.4.2 #ifdef macros	175
9.4.3 #if defined(...) && (AND/OR) defined(...) macro	175
9.4.4 #undef macro	176
9.4.5 #include header files	176
9.4.6 #ifndef macro	176
9.4.7 Macro operators	177
9.4.8 #error macro	180
9.4.9 #pragma compiler options macro	182
9.5 Built-in macros	182
9.6 Library macros	183
9.7 Some useful macros	183
Chapter 10 PROGMEM framework	185
10.1 Basic PROGMEM concepts	186
10.2 The PROGMEM type qualifier	187
10.3 PROGMEM get functions	187
10.4 C-like PROGMEM functions	187
10.5 F() macro	188
10.6 PSTR() macro	189
10.7 PGMP helper macro	190
10.8 PROGMEM no-nos	190
10.9 Using built-in variables and memory requirements	191
10.10 char* string memory requirements and PROGMEM	191
10.11 Array of strings	192

10.12 Storing and retrieving read-only float values	194
10.13 Storing and retrieving read-only struct and class data	194
Chapter 11 Arduino IDE bugs	197
11.1 Undo - ctrl-Z	198
11.2 Segmentation fault	199
11.3 Arduino IDE loses it	200
11.4 Checksum error	200
11.5 Stray '\357' in program	201
11.6 Invalid conversion from char'(*)[4] to uint16_t	201
11.7 Mysterious glitches solved by standardizing development	202
11.8 Sketch's serial port gotcha	203
11.9 No alert on externally modified file	204
11.10 Windows Command Processor, aka DOS box	204
Chapter 12 Gotchas	207
12.1 Why does one make mistakes?	207
12.2 Suggestions to reduce gotchas	209
12.3 Macro gotchas	210
12.3.1 Space between name and parameter open parens	211
12.3.2 Tokenization creates extra spaces	212
12.3.3 Semicolon in macro	212
12.3.4 One too many backslashes	212
12.3.5 Missing backslashes	213
12.3.6 Commenting out part of a macro	213
12.3.7 Function as macro parameter may generate side effects	214
12.3.8 Unsatisfactory macro parameters isolation	215
12.3.9 #ifdef...#endif misplacements	216
12.3.10 Unbalanced #ifdef...#endif pairs	217
12.4 C++ gotchas	218
12.4.1 C++ traps and pitfalls	221
12.4.2 One-line multivariable declarations	223
12.4.3 Mixing numeric types	223
12.4.4 Overflow/underflow during expression evaluation	224
12.4.5 Dereferencing has low precedence	225
12.4.6 Dereferencing a function pointer parameter	226
12.4.7 Redefining a variable or object which already exists	226
12.4.8 Bad pointers - failure to check allocation success	227
12.4.9 Bad pointers - using a pointer directly without assigning memory space	228
12.4.10 Bad pointers - failure to set pointer to zero after free or delete	229
12.4.11 sizeof gotchas	230
12.4.12 new int() vs new int[]	231
12.4.13 Comma instead of semicolon	231

12.4.14	Function call - missing ()	232
12.4.15	Type checking leniency	232
12.4.16	char string concatenation	234
12.4.17	Return values	235
12.4.18	No code after a label	236
12.4.19	Size of an array passed as a function parameter	237
12.4.20	float to uint32_t conversion - problem using pow()	238
12.4.21	Zero-based indexing forgotten and null string termination	239
12.4.22	Default function type	241
12.4.23	Bit-level coding and precedence	241
12.4.24	Function not called	243
12.4.25	Wrong number of parentheses or curly braces	243
12.4.26	Unwanted automatic curly brace	244
12.4.27	Negligent copy/paste leads to bad declarations	245
12.4.28	Memory corruption	245
Chapter 13	Interpreting error messages	249
13.1	Expected initializer before 'xyz'	249
13.2	Expected primary expression before 'char Foo(10, char* msg);'	250
13.3	Expected unqualified-id before '{' token - missing first '\ in macro definition	250
13.4	Expected primary expression before ')' token - forgot macro parameters	251
13.5	Expected primary expression before '{' - label gotcha	251
13.6	Expected ';' before '{' - initializer left in	252
13.7	'Serial' does not name a type - forgot a backslash in macro definition	252
13.8	Stray '\ in program - one too many '\ in macro definitions	253
13.9	Call overloaded 'myFunction' is ambiguous	253
13.10	Multiple types in one declaration	254
13.11	Misleading message following an enum declaration error	255
13.12	Inaccessible member of	255
Chapter 14	Psychological factors	257
14.1	Maslow's pyramid (motivation)	259
14.2	Psychology of computer programming	260
14.3	Cognitive dissonance and EGO	260
14.4	Good and bad habits	261
14.5	How to be an efficient programmer	262
14.6	Understand how your body and mind function	263
14.7	Plan your work offline	264
14.8	Think!	265
14.9	Incremental vs. planned programming	267
14.10	Practical considerations from a psychological perspective	269
14.11	Examples of psychologically induced errors	270
14.12	Key psychological factors	271

Chapter 15 Appendix	275
15.1 PROGMEM framework program	275
15.2 SafeArray class	276
15.3 Pointer arithmetic	280
15.4 AtmelStudio vs. the Arduino IDE - code used or not used	282
15.5 Transmission constraints - 255 not allowed	284
Chapter 16 Bibliography	285
16.1 Bibliography - C/C++ programming	285
16.2 Bibliography - PROGMEM framework	286
16.3 Bibliography - Software Engineering	287
16.4 Bibliography - Regular Expressions (regex)	287
16.5 Bibliography - Awk	287
16.6 Bibliography - Perl	287
16.7 Bibliography - Arduino	288
16.8 Bibliography - AtmelStudio (now MicrochipStudio)	288
16.9 Bibliography - Visual Micro	288
16.10 Bibliography - PlatformIO	288
16.11 Bibliography - Espruino and JavaScript	289
16.12 Bibliography - Hardware-based debugging	289
16.13 Bibliography - Programming psychology	289
A note on the book's source code	291
About the author	293
Index table	295

This page intentionally left blank

List of tables and figures

Table 1.1 - GitHub products for Arduino, RaspberryPi, and ESP32 - 11/19, 12/20 and 10/21	16
Table 5.1 - C numeric types min/max values	54
Table 5.2 - Some common type qualifiers to create derived types	56
Table 5.3 - Table of operators.	65
Table 5.4 - Symbols used as type qualifiers and as operators	66
Table 5.5 - Declaration examples.	79
Table 7.1 - Data packing into bitfields - typical examples	132
Table 7.2 - Date and time packed data - from 8 bytes to 5 bytes.	133
Table 8.1 - Flash, RAM, and EEPROM sizes of Arduino microcontrollers.	154
Table 8.2 - How RAM use evolves.	158
Table 10.1 - Memory requirements using the PROGMEM F() macro .	189
Table 10.2 - Flash memory used with and without the F() macro.	189
Table 15.1 - Min/max values of numbers which exceed 255.	284

This page intentionally left blank

Acknowledgements

This book, *Pragmatic C++ Arduino Programming* and its companion, *Defensive C++ Arduino Programming*, are the result of chance encounters which led me to beekeeping and to create Arduino-based gadgets. Put the two together and, aha! Why not create an *Arduino-based beehive weighing system*. That is how it all got started. And one thing leading to another, I got into writing two books which address the needs of C++ savvy DIY Arduino makers.

I must thank the many who contributed to my getting started on writing these books and continuing it to its ultimate conclusion.

First in line is Daniel T. I am particularly grateful to him since he made me discover Arduino. His electronics advice, despite his being a practicing pediatrics surgeon, hence electronics not being his field at all, contributed immensely towards getting me started with Arduino.

And I thank Christine C., my psychoanalyst, whom I see regularly to express my little travails. She has been an unconditional supporter of my endeavor.

And then there is Charlie H., a practicing physician and fellow airplane builder (RV8). He introduced me to beekeeping - a few visits to his bee yard and I was hooked.

There is of course Xavier M., who purchased my company years back and who has since become a friend. His continued support has contributed to my persevering in this book's endeavor.

My neighbors Martine and Olivier B. continuously supported my endeavors. My special thanks to them. I must say that as the project advanced from milestone to milestone, we celebrated by opening one or two bottles of Champagne - by now, a few cases have gone down our respective esophagus.

As luck would have it, Allison (Olivier's daughter) is an InDesign professional consultant. I am thankful to her as she agreed to create the print and digital ready document. She was patient as she suffered through my unorthodox approach which consisted in creating tags in Word which would be used by an InDesign JavaScript script to produce the finished book (layout, cross references, indices, table of contents, etc.).

I must also thank daughter #1 Giselle, also an author, for her continued support.

Finally, but not least, I owe daughter #2 Pascale special thanks as she patiently proofread both manuscripts, a total of four hundred plus A4 pages of tight letter size text. Since I am an engineer, my thought process, hence sentence construction, tends to be a bit linear (somewhat tedious to read). She managed to smooth things out and put some pep into many of my phrases. And I thank all those others who manifested their support as they patiently heard me out as I described my project.

This page intentionally left blank

Preface

This book, *Pragmatic C++ Arduino Programming*, is the first book of a two-book set, the other being *Defensive C++ Arduino Programming*. They are a by-product of my current beekeeping hobby - it led me to develop an *Arduino-based beehive weighing system*. On the programming side, the Arduino IDE seemed to be the perfect tool: user-friendly and free. I installed it, pulled out my old C++ textbooks (yes, I had written some C++ in the past), ran the *Blink* program on an Arduino Uno, and gradually learned to program Arduino. I managed this at the ridiculously low cost of about \$30 including the hardware. Trying out the examples provided in the IDE and experimenting with small electronic circuits on the Uno were immensely rewarding.

Things did not turn out as easy as I anticipated. I discovered that my C++ skills had gotten a little rusty over time, which led me to do some serious reviewing. I started with the most basic features of the language by doing a "Hello World" to make sure I did not miss anything. I then spent considerable time doing breadboard work on my subsystems: opamps, clock, radio, and GSM. This proved to be time consuming, much more than I first anticipated, but I did manage to get the individual systems to work. I was beset by glitches: problems which occurred occasionally, unable to make them occur systematically. Part of the problem was electrical stability because I was using breadboards. They are fine for as long as you are dealing with DC current; but, as soon as you start doing serial communications, poor connections and capacitive effects corrupt signals. It is what got me into creating soldered prototype boards on top of an Arduino ATmega2560. This combination provided both ample storage space and lots of dynamic memory space (RAM). But I was still getting glitches. The other part of the problem proved to be my programming - it was peppered with errors which I qualify as *traps and pitfalls* and *common programming errors*. These are reviewed extensively in *Gotchas* (page xv).

Let me explain. C++ is deceptively simple. I use the word deceptively because you and I, inexperienced C++ programmers, will be fooled by C's relatively simple syntax. You will inevitably fall into one of many C++ gotchas - an unexpected problem will stop you dead in your tracks and you do not have the slightest clue as to why. C++ is a minefield but do not let this fact scare you - just be careful. You dedicated hours to debugging yet failed to identify the culprit. I have personally been caught by every trap, pitfall, and common programming error C++ could lay along my path.

Gotchas are very real - they will slow you down and weaken your application. If you are to undertake safe programming, i.e., not waste inordinate time finding and eradicating your bugs, you need to have a sound foundation in C++, understand how easily errors can creep in, and organize your code. This is where pragmatic programming comes in, the subject matter of this book.

In the process of reviewing my C++ skills, I found that most textbooks were overkill. By this I mean that they address the needs of the professional C++ programmer and not the needs of the Arduino programmer such as myself, who does it as a hobby. We Arduino programmers need short, practical, clear explanations. Faced with a vocabulary which can become cryptic (lambda, pass by reference, namespace, etc.), learning the language can be tedious. As a newbie Arduino programmer, I found myself overwhelmed by the sheer quantity and depth

of advice as to what, how to, and why. The *C++ Core Guidelines* is more than 500 pages long (see *Bibliography* page xvi); although an excellent and exhaustive work, it is overkill when addressing the needs of an Arduino programmer, and much too cumbersome to be practical. Another example, the excellent Bjarne Stroustrup's *The C++ Programming Language* book (4th edition) is 1347 pages long and covers C++ up to C++11. There is C++17, and more recently, C++20. Bjarne's first edition is 328 pages long - the extra 1000+ pages of his 4th edition illustrate how much the language has evolved and how much more there is to C++.

The typical Arduino programmer needs to master just a small subset of the language. Be pragmatic - learn what you need and be aware as to what you might need later on. This book explains in detail the essential subset and describes most of the remaining features which you will probably never use within the context of Arduino programs.

The core C++ concepts which I believe one needs master are based upon my experience developing my *beehive weighing system* - it got to be big: 35 files, 15,000 lines of code. Unfortunately, I miserably failed to insert comments into my code to document the algorithms. As I progressed on my application, I had a hard time understanding code I had written. This incited me, slowly but surely, to adopt good programming practices: comment code, organize files, develop AtmelStudio/Visual Studio/Arduino IDE interoperability, etc. As my experience and knowledge evolved, the *defensive programming* concept started to materialize. After a while, I had considerable content which led me to write a second book: *Defensive C++ Arduino Programming*, a set of C++ *how to*: getting to know and use AtmelStudio, Visual Studio, Visual Micro, Perl, Awk, regular expressions, and toolboxes (frameworks). The tools I learned how to use and the frameworks I developed helped me improve my productivity and render my Arduino application more compact, fast, maintainable, and robust. *Defensive* means go beyond being good at programming with C++; it means use the right tools and techniques.

Introduction

The two books, *Pragmatic C++ Arduino Programming* and *Defensive C++ Arduino Programming*, are offsprings of my *Arduino-based beehive weighing system* endeavor. In the process of developing it, I had to review my knowledge of C++, all the while taking notes to consolidate my learning. I also developed *frameworks* to better organize my program, more notes; these ultimately morphed into two books.

I had sufficient C++ experience, albeit a touch rusty, to develop my application. I consequently reviewed C++ on an as needed basis. There were many advanced C++ features I did not use but, since I was progressing nicely and since I had no need for these advanced constructs, I concluded that they concerned professional programmers writing large applications. I did ultimately use *operator overloading* and simple *inheritance*, got to use the *& reference qualifier*, and created an *exception handling* like mechanism for error handling.

Lots of C++ books have been published. So why another one? The answer is that I found no book which addresses the needs of the already savvy self-taught C++ Arduino programmer. This book is meant to be a *pragmatic primer* of C++ features an Arduino programmer will use. It separates the *basic features* we should know from the *advanced features* none of us Arduino programmers are liable to use. It is not a reference work for professional programmers, nor a textbook for a course. Its aim is to be a *pragmatic presentation* of C++ to help Arduino developers improve their productivity and enhance their understanding of the language's features. An additional goal is to alert the C++ programmer that the language is treacherous - its simplicity is deceptive. You, as I did, will spend far more time debugging than developing the application. This book covers typical errors one might make.

The title of this first book contains the word *pragmatic*. Being *pragmatic* implies that practicality dictates "What one needs to master". Arduino developers (Atmel, ESP32, and others) need to master basic features and have a working knowledge of advanced features. Behind the specialized terms lie simple but subtle concepts (pointers, pass by reference, classes, etc.). Furthermore, over the years, the language has integrated ever more complex mechanisms (lambda functions, inlining, concurrency, multitasking, exception handling, regular expressions, etc.). Although most of us Arduino developers do not need to use these advanced features, they are listed and described in *What one needs to be aware of* (page 1).

I wrote this book with an eye towards practicality. Its content is broken up into chapters, as follows:

- "*Chapter 1 Arduino and C++*" (page 1) covers what Arduino is, where it comes from, why C++, and what it is good for (pros and cons), microcontrollers, programming languages, and language/chip combinations.
- "*Chapter 2 Arduino IDE*" (page 1) introduces the Arduino development tools: the Arduino IDE, the Arduino C++ editor, the GNU toolchain (preprocessor, compiler; linker, make utility, avrdude, bootloader).
- "*Chapter 3 Other IDEs*" (page 1) introduces AtmelStudio (MicrochipStudio), Visual Micro for MicrochipStudio (AtmelStudio), Visual Micro for Visual Studio (Microsoft), VS Code, PlatformIO, Code::Blocks. It also introduces some visual development tools and using AI (ChatGPT) to kick start specific development needs.

- "*Chapter 4 What one needs to master*" (page 2) - C++ building-blocks (identifiers, types, operators, etc.) and the mechanics to assemble these into functional features (functions, scope, type checking, polymorphism, etc.) are what one needs to master.
- "*Chapter 5 C++ building-blocks*" (page 2) are items one works with: variables, operators, control flow statements, functions, classes, etc.
- "*Chapter 6 C++ mechanics*" (page 2) are rules which govern using the *C++ building-blocks* - they should be well understood. You should know how the *pass by reference* mechanism works, organize bitfields, and understand how operator *precedence* affects a statement's evaluation, and more.
- "*Chapter 7 What one needs to be aware of*" (page 2) - These are the C++ concepts the Arduino developer should be aware of but would probably not use.
- "*Chapter 8 Memory management*" (page 2) - Having enough memory throughout an application's life cycle is crucial to its performing reliably. Memory allocations and function calls consume memory; the programmer should at all times ensure that the application's needs are met.
- "*Chapter 9 Macros*" (page 2) *are a unique feature of C++* - They are handled by the preprocessor, a text replacement, conditional inclusion, and file inclusion utility which preprocesses the source file before passing it on to the compiler. They grant the programmer flexibility not available in other programming languages.
- "*Chapter 10 PROGMEM framework*" (page 2) - Arduino provides facilities to store read-only variables and constants in flash memory, thereby providing the possibility of saving considerable RAM space.
- "*Chapter 11 Arduino IDE bugs*" (page 2) - Nobody is perfect. The Arduino IDE, the editor, compiler, and linker in particular, manifest some annoying bugs which finally pushed me into using AtmelStudio (and later Visual Studio/Visual Micro) as my main Arduino programming tool.
- "*Chapter 12 Gotchas*" (page 2) - Since C++ programming is akin to walking through a minefield, a roadmap of mines (gotchas) should be included in a C++ programmer's training. There are two kinds of *gotchas*: *macro gotchas* and *C++ gotchas*. Given that the preprocessor is a rather unsophisticated text find and replace processing tool, it can be a source of surprising runtime problems, particularly since the preprocessor does not generally produce error messages. Both *macro gotchas* and *C++ gotchas* are extensively expanded upon.
- "*Chapter 13 Interpreting error messages*" (page 2) - It is unfortunate that compiler designers remain entrenched in their highly specialized lingo - some error messages are downright ludicrous.
- "*Chapter 14 Psychological factors*" (page 2) - Our minds are complex machines. Be careful with your subconscious - it will make you do things which are not good. Laziness, persisting down an erroneous path, not preparing sufficiently before undertaking a task - all these can lead to excessive debugging, duplicate work, and unnecessarily complex algorithms.
- "*Chapter 15 Appendix*" (page 2) presents details on setting up a SafeArray class which overloads the index [] operator to check on possible out-of-bounds array indexing conditions, doing pointer arithmetic, defining code used or not used, and a how-to

avoid using 255 values during radio transmission.

- "Chapter 16 Bibliography" (page 3) covers C++ books, YouTube videos, documents, and links I found relevant.
- *A note on the book's source code* (page 3) details how to get the book's source code from *md-dsl.fr* - MIT Open Software License.
- *About the author* (page 3) provides insight as to why I wrote this book and its companion: *Defensive C++ Arduino Programming*.
- *Index table* (page 3) - Since most technical books are reference works, a comprehensive index table is a must. It is frustrating to open a book, look for something which is surely in the book, yet it is not in the index table. The reader should find, via the index table, just about everything that is in the book.

If your Arduino program is thousands of lines of code, having a good understanding of C++ is not good enough. You need to be pragmatic, which means *know and understand enough to get the job done correctly*. You also need to *apply defensive programming techniques*: work with good tools, understand how to organize workflow, avoid reinventing the wheel. You need to use frameworks and adopt a professional developer mindset. The companion book, *Defensive C++ Arduino Programming*, meets these needs; it should help you improve your productivity and render your application more robust.

This page intentionally left blank

Chapter 1

Arduino and C++

Arduino is a fantastic tool. It grants hobbyists access to microcontroller development at a ridiculously low cost. As of 2022 it costs a mere \$30 plus a little more for the electronic components. On the software side, the Arduino IDE is incredibly easy to use. Furthermore, its learning curve is short, and it is free. With this in mind, you should seriously consider a contribution to the Arduino group if you are to use Arduino extensively. In a word, using Arduino is a win-win. Get an Arduino board, get a breadboard and a few components, install the Arduino IDE, and voilà, you are ready to go. Add more components, bring in more functionality, and you will find that your initial Arduino endeavor is turning into a major project - it can be a lot of fun.

Before continuing, a word on "Why C++?". Microcontrollers are expensive; consequently, using the smallest possible microcontroller to meet the needs of a given job is a top-level design imperative. This is why practically all programs which need to be as compact and fast as possible are written in C++. There is just no other language which can beat it aside from assembler but using it is prohibitively costly from a development, debugging, and testing perspective. Choosing a chip/language combination is expanded upon in *Which chip/language combination?* (page 5). And now, back to Arduino.

Creating a *beehive weighing system* is what got me into learning what I needed to know on Arduino and C++. My project (as well as most projects) may be summarized as follows:

- *Proof of concept* - I initially worked on breadboards during the proof of concept of individual components: strain gages connected to opamps, optocouplers, voltage regulators, radio communications, sending/receiving SMSs, and clocks.
- *Prototyping* - I migrated from breadboard Arduino UNO to soldered prototype boards on top of ATmega2560 boards.
- *Final product* - I created my own stand-alone Atmel based PCB directly programmable from my PC.
- *Programming Arduino* - As I worked on the hardware, I progressively learned how to use the Arduino IDE and the Arduino specific C++ mechanisms to develop the program which would drive the *beehive weighing system*.

At some point during the development phase, I looked at AtmelStudio. What I initially discovered prompted me to conclude that the learning curve was quite steep and that importing an Arduino project was complicated. I therefore continued with the Arduino IDE despite the occasional *segmentation fault* or *mess up my source code undo (ctrl-Z)*. Then, one day, Atmel released AtmelStudio V7, the all-important new feature being its ability to create AtmelStudio projects directly from Arduino *.ino* sketch files. This new feature, plus the fact that I was fed up with *ctrl-Z* messing up my source code, and the *segmentation fault*, got me to try AtmelStudio. So, I

buckled down and dedicated time to AtmelStudio - its learning curve proved to be surprisingly short. I tried importing a small sketch and was successful. Wow! It was easy and worked well. Then I tried it on my large *beehive weighing system* Arduino program. To my surprise, importing my 35 files/15000 lines of code was quick and flawless. I gradually discovered how a real professional IDE could make coding a lot less stressful. It was a revelation - I was in programmer heaven. From then on, my programming experience changed for the better. The AtmelStudio learning curve turned out not to be steep at all - I was operational in no time. What is more, I developed a technique to easily switch from the Arduino IDE to AtmelStudio and vice versa with little effort. It was just a matter of activating one of two macros (`#define ATMEL_STUDIO` or `#define ARDUINO_IDE`). You will find full details on the *AtmelStudio/Arduino IDE interoperability framework* in the companion book *Defensive C++ Arduino Programming*.

Since there were other possible IDEs, I looked at PlatformIO, Visual Studio, Visual Micro, and Code::Blocks, just in case they proved to be an improvement over AtmelStudio. PlatformIO is a formidable tool in that it supports Arduino, ESP32, and other microcontrollers. It also supports hardware-based debugging (in a limited way for Arduino), and teamwork configuration management via external tools. As for Code::Blocks, it looks somewhat like AtmelStudio, but the similarity ends there. I was not able to import an existing Arduino project into it. It might be possible, but it looks like being a real hassle. I consequently stopped testing it. This being said, there might be a solution which requires doing a little research. I shall conclude by saying that I found AtmelStudio (and later Visual Studio/Visual Micro) user-friendly and powerful enough for my requirements. You will find short descriptions of these tools in their respective short introductory chapters. See *AtmelStudio* (page 6), *Visual Micro for MicrochipStudio (AtmelStudio)* (page 6), *Visual Micro for Microsoft Visual Studio* (page 6), *PlatformIO* (page 6), and *Code::Blocks* (page 6). These five tools are described in more detail in the companion book *Defensive C++ Arduino Programming* (see *Bibliography* page 6).

But walk before you run. For a more detailed presentation of the Arduino solution see *Arduino IDE* (page 6). This chapter, *Arduino and C++*, covers the following themes:

- *A short history of C and C++* (page 6) - In the old days, computer manufacturers like IBM (mainframes), Digital Equipment (VAX computers), and others created their own operating systems and programming languages. This means that once a business decided upon a specific computer, it got locked in as its applications could not be ported to other computers. The need for a portable operating system and programming language combination became pressing. MIT worked on such a system (*multics*) - Bell Labs did more work on it to finally produce Unix and the C language. Later on, Bjarne Stroustrup, while working on his PhD thesis, came up with an object-oriented extension to the C language; he later named C++.
- *C++ programming* (page 6) - As Unix proliferated, mainly on large computers (PCs and Macs were relegated to being desktop computers), the C language became a standard for developing mission-critical applications. It however quickly became apparent that C programming (procedural programming) was a painstaking non-programmer-friendly way of developing applications. The price to pay was complexity. The aha moment of the industry was "Let programmers create their own user-defined types" (dixit Bjarne Stroustrup) and this is how C was enhanced to being C++.
- *Microcontrollers* (page 6) and the CPU which drives your PC are really one and the same electronic device type in that they are conceived to be programmed, i.e., intelligence

can be programmed into them. However, the term microcontrollers refers to embedded devices such as in cash registers, in my *beehive weighing system*, etc. For small DIY projects, Arduino (Atmel 8-bit architectures) is an excellent choice.

- *Programming languages* (page 7) - When choosing a programming language, a host of considerations come into play: programming ease, application speed and compactness, programming languages you already know, etc. Most of the time, the choice is a compromise between several opposing criteria. C++ excels when application speed and compactness are crucial. It is however a time-consuming language to program with as compared to Python.
- *Which chip/language combination?* (page 7) - Choosing a programming language and a chip to build an application on are interrelated. It all depends upon what you know and your priorities. The three main chips one would consider for small DIY projects are Arduino (Atmel 8-bit chips), RaspberryPi, and ESP32. And then you would short list C++ and Python as the programming languages. That is six chip/language combinations. Which one to opt for? I personally chose Arduino/C++ mainly because I wanted the smallest, fastest applications, even though this choice would cost me extra development time.
- *What defines C++* (page 7) - Having settled on C++ as the programming language, the questions are: What is it? What does it look like? How does one use it? What can it do for you? One should start by looking at C, how it differs from other programming languages by providing the programmer features not generally available elsewhere such as a preprocessor (macros), direct work on memory addresses, bit-level manipulations, simple syntax, rich set of operators, etc. Once you have gotten a grasp of the C language, add object-oriented programming (*user-defined types*) and voilà, you understand what C++ is.

The pages which follow cover the above, the nitty-gritty of C++ programming (see *What one needs to master* page 7).

1.1 A short history of C and C++

In the old days, computers were programmed in machine language by way of an assembler. It transformed source level machine instructions into machine code - it assembled machine instructions, hence its name. Such programming proved to be extremely time consuming but there was no other way. Let me illustrate machine language.

Suppose you wanted to do a simple addition such as

```
c = a + b
```

The above is classic source code in just about any programming language. The machine language equivalent would look like this:

```
Transfer content of memory location A to register 1
Transfer content of memory location B to register 2
Add contents of register 1 to register 2
Transfer content of register 2 to memory location C
```

One line in a higher-level language replaces four lines of machine language code, making it considerably more understandable.

Writing machine language code is unbelievably time consuming, difficult to read, error prone, and a debugging nightmare. These impediments were resolved with the development of

higher-level programming languages to abstract away the architecture of the microcontroller. Unfortunately, at the time, each computer manufacturer developed its own higher level programming language - hence there was no portability across computer manufacturers.

In the mid-60s, M.I.T. started developing a multitasking operating system named *multics*. Taking *multics* as a model, Bell Labs created Unix and the C language to develop it with - C and Unix were meant to be portable across machines. The initial Unix operating system was created along with an initial minimalist C compiler. C's initial characteristics included core language features plus standard library components. Variables, return values, etc. had to be declared as to type so that the compiler could allocate space. Individual program modules were compiled and linked together to create an executable file. This opened the way to precompiled library modules to encapsulate predefined functions.

From then on, Unix developers used the C language to both extend Unix and to extend the C language and compiler - the C language was used to extend the C compiler which was then used to extend the C language; it was meant to be both programmer friendly and general purpose. Its basic features were flexible built-in numeric types, a rich set of operators to manipulate the data, facilities to create complex numeric types, control flow statements, and functions to encapsulate logic. It became a procedural programming language capable of creating compact code which ran efficiently on many computers.

Then came object-oriented programming. Before exploring how and why it came about, you need to understand a fundamental concept: simplicity. It is one of the keys to achieve programmer productivity and create efficient, robust applications. Let me explain.

Procedural programming languages evolved but they still lacked features to facilitate top-down programming (program the way one thinks). The question raised is: *What is the fundamental feature a language should have so that the programmer can write clear, readable, maintainable code?* Before reading on, think about this and try to answer the question. I personally was surprised by the answer: *Make it so that programmers could create their own types*. This is what Bjarne Stroustrup said in a recent interview. User-defined types do away with detail/clutter. It enables *top-down* work - programming becomes cleaner. Start by laying down the overall architecture and later on, fill in the details (*bottom-up programming*).

Procedural programming languages do sort of support *user-defined types*. Sure, you can create a struct which in a way is a new type but that will not take you far. A C struct does not have functions - work on its content must be handled by global functions. Imagine trying to build entities such as boxes, spheres, and cylinders in C. You can but the logic will be spread out over individual global functions such as

```
float VolumeSphere(struct _sphere);  
float VolumeBox (struct _box).
```

Furthermore, logic is limited since there is no *user type* which could abstract away all the different physical containers one might want to deal with.

User-defined types became the means whereby programmers could simplify their code. Algol (Algorithmic Oriented Language) is the first language to implement this concept. Algol led to Simula which Bjarne Stroustrup used as a starting point to create a C with classes language for his PhD thesis, which he later named C++. However, he was not the only one working on object-oriented programming. Other object-oriented programming languages came into being in the 80s (e.g. Smalltalk and Objective C). Encapsulation enabled the programmer to enclose related functionalities inside an entity (call it a *class*), a class being a programming feature

which contained both data and functions to act upon the class's data.

C++ became generally available in 1985. It extends C into the object-oriented programming realm. It is not a better C - it is a C with classes, a C with *user definable types*. It is C++, an object-oriented programming (OOP) language based on user definable `class` definitions which contain functions (methods) to work on the class's data. It is worth noting that an Arduino programmer could write his/her program entirely in C - functions and variables only. However, a sprinkling of C++ here and there will simplify code, clarify program logic, assure better maintainability, reduce development time, and bring other desirable results.

The key points to understand are:

- *Good code* needs to be simple - simplicity makes life easier for the programmer and gives the compiler a chance to generate efficient (fast and compact) machine code.
- To achieve *simplicity*, the programmer must be able to create his/her own types. This can be achieved via encapsulation. It is the process of enclosing data and functionalities inside an entity - call it a `struct` or a `class`. `class` specific functions manipulate the `class`'s data. The programmer may thus define his/her own types and use them the same way as using `ints`, `chars`, `floats`, etc. See *class and struct* (page 9).
- Features such as *inheritance* and *polymorphism* enhance *user-defined types* (`classes`). These concepts are covered further on.
- *Strong type checking* (ex. when passing parameters to functions) became an added feature of C++. The compiler will trigger an error if you pass a `char*` instead of a `float`, thus a safer language than the original C language.
- *Having understood the above* (the big picture), the rest is details.

C++ has evolved considerably since 1985. New features which address the needs of highly skilled professional programmers have been added, two of which, concurrency and regular expressions (`regex`), are particularly noteworthy. The current (May 2023) standard is C++20 - C++23 is in preview phase (see Wikipedia C++)

One last item merits attention: Why the `++` in C++? Bjarne Stroustrup considered that the `++` operator, which means increment, was an accurate presentation of his enhancements to the C language. C++ is C incremented with user-defined types (`class`). Furthermore, `+` stands for positive and `++` is doubly so. The `++` in C++ could also be viewed as a marketing gimmick.

1.2 C++ programming

When choosing a programming language, it is often a tossup between programming ease and other criteria. When coding for microcontrollers, code compactness is all important because the smaller the code, the smaller the microcontroller, the smaller the cost.

If programming ease were to be the most important criterium, Python would prevail over C++. If compactness were to far outweigh programming ease, the most efficient language would be machine code (assembler), but this proves highly impractical as development time becomes prohibitive. C++ programming is a good compromise in that it is programmer friendly and generates compact code. It has been developed to enable compilers to be close to the underlying microcontroller's architecture, yet it offers the programmer a syntax that is easy to implement with sufficient built-in features to accomplish the most sophisticated programming tasks. If some machine code for critical program sections were required, the programmer could resort to inlining assembler code within the C++ code.

From a code compactness perspective, C++ differs from Python in that C++ is a compiled language, whereas Python is an interpreted language. C++ does not need an interpreter to be uploaded into the microcontroller to get the program to run, such as Python requires. C++ source code is converted into a machine executable file and then uploaded into the microcontroller. This is why C++ code is much more compact than Python's and faster.

Historically, C, Basic, Fortran, Pascal, and other 3rd generation programming languages provided the programmer with fundamental mechanisms common to most programming languages: bottom-up programming. These include algebraic expressions, variables and arrays, logical expressions, program control flow mechanisms, and mechanisms to encapsulate data (struct in C, record in Fortran, ...) and logic (functions, methods, subroutines, procedures, ...). Basic has evolved into Visual Basic and is extensively used for Microsoft Word, Excel, etc. related development. Pascal has morphed into Delphi which has its own following. Fortran is still much used by engineers despite conventional wisdom which says that it is dead. And C has evolved into C++ (object-oriented programming OOP). Somewhere along the line, Microsoft invented C# (C-sharp), similar to C++, specifically tailored for Microsoft .NET development.

C differs from other 3rd generation programming languages in that it was designed for the Unix operating system. It was to be small yet provide the programmer with the user friendliness of 3rd generation programming languages (Basic, Pascal, Fortran, etc.). Designed for professional programmers, it assumes that the programmer knows what he/she is doing. C does not protect programmers from themselves the way Fortran does. Because of this fundamental design feature, C requires that the programmer be good otherwise, gotchas will take their toll: array overruns, memory corruption due to unallocated pointers, improper numeric type usage, stack overflow, and more. These are extensively described in *Gotchas* (page 10).

It is well worth repeating that C++ is deceptively simple. Do not stop with learning how to write individual lines of code. You should understand C++'s underlying concepts. This is what this book is all about - it explains how to use the language. Adhering to the precepts presented will increase your productivity by helping you avoid C++'s traps, pitfalls, and programming errors. This book is meant to be a reference of C++ features you are likely to use. It also covers features which you are unlikely to use but which you should be aware of. For a more complete coverage of C++ features, you may consult GCC's excellent manual (gnu.org/software/gnu-c-manual/gnu-c-manual.pdf). It is clear, concise, to the point. You could also read Bjarne Stroustrup's *The C++ Programming Language* book, 4th edition as it is the most complete book on C++. Do not start at page 1 and read on - skim through it, get a general understanding of its contents so that later on you would know where to go to deepen your understanding of some specific C++ feature. It is a big book, 1347 pages.

1.3 Microcontrollers

Microcontrollers differ tremendously from one another. The question is: Which microcontroller should one choose for one's project? Below are a few non exhaustive features which characterize a microcontroller:

- *Register size* - Microcontrollers can have 8-, 32- or 64-bit registers. This means that a unit of storage may be small or large. Types such as bytes (8 bits), integers (16 bits), long integers (32 bits), etc. are generally supported, they do not depend on the register size. This being said, it remains a fact that the larger the register size, the faster the

application.

- *Speed* - Are we dealing with mega or gigahertz? This affects the application's speed, which may be critical when doing signal processing. The microcontroller's clock rate and register size duo determine speed.
- *Memory* - Microcontrollers generally have three types of memory: *dynamic* (RAM), *flash memory* (where the application, the *bootloader*, and PROGMEM data reside), and EEPROM (permanent storage available to the application). See *Arduino memory pools* (page 11).
- *Ports* - Does the microcontroller provide digital and analog ports? Digital is one of two voltages: HIGH or LOW (e.g. +5V or 0); Analog is continuous between HIGH and LOW.
- *Timers* - Can the microcontroller supply time? With what resolution?
- *Clocks* - Can the microcontroller keep date/time of day? With what precision?
- *Interrupts* - Does the microcontroller support interrupts? How? Both hardware and software interrupts?
- *Communications protocols* - Does the microcontroller support WiFi, Bluetooth, Serial, etc.?

Here are other considerations when choosing a microcontroller.

- *Programming languages* - What languages can the microcontroller be programmed with? C++ only? Python?
- *Operating system* - Does the application run directly once it is loaded? Or does it require an interpreter? This has an impact on memory requirements since the operating system or interpreter require memory.
- *Multitasking and concurrency* - Multitasking means run two or more tasks not necessarily concurrently - the tasks could run in a round-robin fashion such as do one thing, stop doing the one thing and do another for a while then come back to do the one thing. Concurrency requires two or more processors so that tasks can run in parallel. Multitasking could be done with one processor only, such as happens whenever an interrupt triggers an ISR (interrupt service routine).
- *Costs* is often an important issue. Is the microcontroller available as a standalone or only as part of a board? At what cost? It may be desirable to use boards such as Arduino and RaspberryPi for development purposes, then migrate to custom made PCBs in which the microcontroller sits alone. Is this feasible?

Many other features characterize a microcontroller. Analyzing the most common ones in detail is beyond the scope of this book. In my humble opinion, the three most used microcontrollers for DIY projects are Atmel microcontrollers (Arduino - available as standalone), RaspberryPi and ESP32 (these two mainly available on boards, hence considerably more expensive).

- *Arduino* - The term Arduino refers to a family of boards originally based on 8-bit Atmel microcontrollers, later extended to other microcontrollers along with an *integrated development environment* IDE. The creators of Arduino designed a software tools/microcontroller boards combination that is technically referred to as cross-development. You create code on a computer and then upload the executable into a microcontroller. There is usually no operating system. A small application (*bootloader*) transfers control to the application which resides in flash memory. Instructions are then transferred

from flash memory into the microcontroller's registers, one at a time, for execution. This feature, combined with a low-level language such as C, maximizes the use of the microcontroller.

- *RaspberryPi* is a family of boards which can be programmed in Python, C++, and other languages. They interface with an operating system thereby enabling applications to become full featured miniature computers, complete with keyboard, screen, and SD card. They can be used to create terminals of all sorts such as cash registers, payment terminals, and facilities access systems. However, RaspberryPi may be overkill for small IoT devices and DIY devices, and it costs more than Arduino.
- *ESP32* - Power characterizes the *ESP32* since it is a 32-bit processor. It is in many ways a super Arduino. It can even be programmed via the Arduino IDE. Because it has two cores, it can undertake concurrency, i.e., run two tasks in parallel. Furthermore, it supports WiFi and Bluetooth.

There are many more choices when choosing microcontrollers, it all depends upon what the application needs to do. But the three listed above (Arduino, RaspberryPi, and ESP32) will fulfill a majority of DIY project needs.

1.4 Programming languages

For the sake of completeness, I have listed below short descriptions and features of leading programming languages, namely: C/C++, C# (C-sharp), Python, HTML, JavaScript, VBA, PHP, Go, Delphi, Java, Fortran - forgive me if I left out your favorite programming language:

- *C++* is powerful in that it generates small fast code. It has a preprocessor which turns out to be priceless (more on that later) and there are lots of C++ libraries. Practically all major Open-Source Software is written in C++. It has frameworks for multitasking (task parallelization) and more. But it comes with a cost because it is fairly complex and you are constantly introducing programming errors (inadvertently shooting yourself in the foot), which sometimes takes forever to resolve. I should venture to say that programming in C++ takes 3 to 5 times more time than programming the same logic in other programming languages unless, of course, you are an expert seasoned C++ programmer. C++'s additional programming time burden is offset by the application being considerably more compact than had it been written with another language. This reduces the hardware cost. Its faster execution speed could be another benefit. This fact alone is all important when undertaking signal processing. If you are dealing with events in the millisecond range, Python and other programming languages will not do. Same with image processing, if you are to process a large number of data points, in the megas or more, speed is required.
- *C# (C-sharp)* is a Microsoft programming language tied to the .NET Framework. It is an OOP language similar to C++ and Java. It provides useful high-level features not included in C++ such as array bounds checking, detect attempts using uninitialized variables, static garbage collection, etc. C# can be used for embedded development via the Microsoft .NET Micro Framework. The following is the description I found in the *old.dotnetfoundation.org* site: "The Microsoft .NET Micro Framework is an open source platform that enables you to manage C# applications for source constrained embedded devices". It is fully integrated within Visual Studio. I have not tried it because I do not

intend to learn C# programming at this time; but it is good to know that it exists.

- *Python* is an interpreted language - its learning curve is shorter than C++'s. It runs on RaspberryPi therefore, if that is your platform, why not? It has a built-in debugging framework which simplifies development. But Python is slower, and the code is larger because it is an interpreted language. Source code statements are read into the interpreter one statement at a time and executed. This means that the target microcontroller must load the Python engine and source code, thereby occupying precious RAM. It has other drawbacks such as neither being adapted to graphics nor to database management nor to multitasking. If you can live with these limitations, then program in Python. You will get your application up and running sooner than doing so in C++.
- *HTML (Hypertext Markup Language)* is what makes Web pages look the way they do. It can be enhanced with Cascading Style Sheets (CSS) and with JavaScript for more sophisticated rendering.
- *Java* - The idea behind Java is "Write once. Run anywhere!". Java is an interpreted language as opposed to being a compiled machine code language. A Java compiler transforms source code into an intermediate code (p-code) which is then interpreted and run by a Java virtual machine which sits in the computer running the code. It is object-oriented much like C++. It is widely used. The Arduino IDE V1, for example, is written in Java. At this juncture, I know of no Java virtual machines for the common microcontrollers.
- *JavaScript* programming was originally developed for client-side Web development. It is the logic behind the Web pages. One would consequently think that it is not an embedded development programming language - this is a bad assumption. I came across JavaScript being used for Espruino development (See *Bibliography* page 13). JavaScript is also extensively used in Adobe products (InDesign, Photoshop, etc.) to automate tasks. Create a script in JavaScript and run it instead of repeatedly opening dialog boxes to do work.
- *PHP* applications sit on Web servers to generate custom Web pages. For example, a hotel reservation system receives input from the user - the server receives the request, processes it, and sends an HTML and JavaScript code response generated by a server-side PHP program.
- *VBA (Visual Basic for Applications)* - This Microsoft specific programming language enables automating tasks inside and between Office documents. If you have Word, Excel, PowerPoint, or other Microsoft tools, you have VBA. It is a full-fledged programming environment which includes most of the features of a good IDE: syntax sensitive programmer editor, interactive debugger, real time variables display, watch window, function calls, and more.
- The *Go programming language* is recent. A C++ programmer will be immediately at ease with its syntax. Take a tour of the language - it contains lots of interesting built-in features not available in C++, which makes it enticing. It even supports concurrency (concurrency means simultaneous multitasking - parallel processing). This can dramatically improve an application's speed if the microcontroller is multicore, and the operating system or application supports multitasking. The Atmel microcontrollers used

on Arduino being single core, concurrency is not an option.

- *Delphi*, aka object Pascal, is today's successor of the ubiquitous, so popular, Turbo Pascal written by Philip Kahn back in the late 80s. One could say that C++ and C# meet the needs of professional programmers; Fortran meets the needs of engineers; Delphi meets the needs of general programming. It features rapid visual components drag and drop application development. As for Delphi for Arduino, I did find some Delphi references for creating Arduino applications; however, the process seemed somewhat complicated. At some point in time (2017), the big Delphi news was *Visualino* (page 14), a rapid Arduino drag and drop development environment built with Delphi; but this product seems to have been dropped because it has now been six years that no work has been done on it. My impression is that one would be better off using C++ and the Arduino IDE or some other tool. But if Delphi is the only programming tool you know, give it try.
- *Fortran* - But isn't it a dead language??? No! Large industrial firms use it extensively to this day. Intel proposes a Fortran compiler which it supports just as actively as its C++ compiler. Fortran means Formula Translation - it addresses the needs of engineers. C++ addresses the needs of professional programmers who need to write fast, compact code. C++ assumes that programmers know exactly what they are doing. This is not the case for Fortran - it assumes that programmers should be protected from mistakes they might make. For example, in Fortran, array overruns cause a runtime error. In contrast, C++ allows programmers to index an array way outside the array's limits - the program will continue running, and eventually crash long after the damage was done.

The question remains: Which programming language should be chosen for microcontroller development? Python, C++, or some other language? You could use Python on Raspberry or other microcontrollers, but these carry a cost. The generated program is considerably bigger and slower than the equivalent program written in C++. If the program were to be small relative to the microcontroller, why not go for Python, it is an easier language to work with, but you may fall into a trap. As your program grows in size and complexity, and memory usage becomes an issue, you might be forced into a serious rethink. In a nutshell, C++ is the programming language of choice to bring hardware costs down. The smaller the program, the smaller the microcontroller to do the job, the lower the cost.

1.5 Which chip/language combination?

I have introduced C and C++ because they have been all-important in embedded microcontroller application developments. Although microcontroller C++ development has been historically reserved for professionals, the past 10 years has seen a proliferation of new chips and programming environments which has brought both the learning curve and cost down. Today, you may choose one of several microcontrollers and languages to program them with. You will be faced with several options, such as Arduino, RaspberryPi, ESP32, WiPy, Espruino and others, and two mainstream programming languages: C++ and Python.

There are other programming languages (short review above) but, for microcontroller development, C++ and Python constitute the short list. Choosing the microcontroller first and then choosing the programming language or vice versa poses a dilemma. If you choose C++, you can opt for Arduino development and other microcontrollers. If you choose Python or other languages, Arduino may prove problematic. It all hinges on the application's

requirements and what you are comfortable with. For example, the Arduino IDE can output machine code from C++ for a limited set of microcontrollers. MicroPython has interfaces for more microcontrollers (RaspberryPi, WiPy, Espruino boards, ESP32, ESP8266, and there is some support for the Arduino Atmel series).

The question remains: Which chip/language combination? Here are some considerations:

- *What you already know* - You may be a seasoned Python programmer and learning another language may prove to be a daunting task which you do not want to embark on. You may want to choose a microcontroller compatible with what you know.
- *Hardware costs* ultimately depend on the microcontroller choice and size, but it is a given that the smaller the microcontroller, the lower the cost.
- *Microcontroller features* - You may require features available on certain microcontrollers only, such as concurrency, multitasking, multiple hardware serial ports, digital and analog ports, power output, low power consumption (sleep and active), clock, interrupts, memory, etc.
- *Language features* - Your gizmo may require specific features easily implemented by some languages and not by others. Concurrency, supported by the Go language and recently by C++, come to mind. In general, most languages support the same features - what you can do with one, you can do with another, which means that language features are generally not an issue.
- *Development time* - This can be an issue as you will certainly spend considerably more time writing and debugging C++ programs than Python programs.
- *Devices, sensors, hardware support* - Your gizmo may interface with a variety of sensors. Your search in GitHub for 3rd party libraries which support your devices may dictate your microcontroller/language choice. If you do not know what GitHub is, do yourself a favor, go to github.com - you will discover archives of development software to meet just about every imaginable need.
- *Development environment* - Is there a development environment for your microcontroller/language combination? Is it user-friendly? Is it robust and reliable? Does it generate good code?
- *Futureproofing your knowhow* - Make sure that the language and microcontroller combination you choose is evolving and that it is experiencing increasing adoption.
- *Employment* - Why not choose a microcontroller/language combination which has real employment opportunities?

When deciding upon a microcontroller/language combination, each of the above will carry some weight. In my case, I chose Arduino with C++ due to the following:

- *Devices, sensors, hardware support* for the components I needed were available (GSM board, DS3231 clock, DTH22 temperature/humidity sensor).
- *Arduino costs* (board and microcontrollers) being ridiculously low could not be ignored. Furthermore, alternative Arduino microcontrollers (ATmega2560, ATmega328P, Nano, and others) enabled me to adapt the microcontroller to the application's requirements. Low hardware costs were an important factor.
- *The Arduino IDE is user-friendly* and had just enough features for me to get started. It is only later on, as I honed my skills, that I switched to AtmelStudio (and later Visual

Studio/Visual Micro) - it improved my productivity tremendously. See *Other IDEs* (page 16) and the companion book *Defensive C++ Arduino Programming* which fully describes using them.

- *Current knowhow* - I have programmed in C++. The learning curve was consequently not a factor - I just needed some refreshing. I did consider using Python since I should be more productive, but since C++ created considerably more compact (also faster) code and Python support for Arduino at the time seemed to be experimental - choosing Arduino with C++ was a no brainer.

Arduino's ubiquity was an important reason for my choosing it. The likelihood that it might become obsolete seemed far off in the distant future - I qualify it as being futureproof. There are more than 200,000 Arduino libraries which cover just about any device a developer might need. Arduino being the number one choice among possible microcontrollers (RaspberryPi, ESP32, and others), libraries get created for Arduino first, then progressively for other microcontrollers. Searching GitHub with the keywords Arduino, RaspberryPi, and ESP32 yielded the following:

GitHub search	Nov. 2019	Dec. 2020	% increase	Oct 2021	% increase
Arduino	159706	194839	22%	221844	14%
RaspberryPi	12682	15607	23%	17821	11%
ESP32	9982	18497	85%	26720	14%

Table 1.1 - GitHub products for Arduino, RaspberryPi, and ESP32 - 11/19, 12/20 and 10/21

My research of GitHub libraries yielded interesting results. The most immediate conclusion is that there is from five to ten times more content for Arduino than for Raspberry Pi and for ESP32. Furthermore, over a one year's span (2019 to 2021), ESP32 GitHub tools increased by 85%, 15% the following year. The first year witnessed a lot of new stuff for it as it was a new microcontroller. As for Arduino and RaspberryPi, they increased by 14% and 11% respectively which reflects a mature environment.

A note on the book's source code

Frameworks source code: You may download the complete *as is* source code from md-dsl.fr/c-arduino-programming, modify code to your heart's content, and use it free of charge at your own risk on a non-commercial or commercial basis. It is subject to an MIT open-source type license agreement with some restrictions, as follows:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to incorporate it inside of an executable or other machine language component without restriction for personal or commercial use. The Software may not be redistributed as source code or in any recognizable human readable form in any form whatsoever for any use whatsoever.

The Software is the property of Michèle Delsol (France), copyright 2023, USA and international. For any questions concerning use of the software contact Michèle Delsol at CPParduino@md-dsl.fr.

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Note that some frameworks have benefited from extensive development work, and they are quite solid; others are in their infancy and may be buggy.

You will find, throughout this book, code snippets, classes, functions... to illustrate how C++ works. They cover a lot of material. Some of them are short and to the point; others are longer and can be more or less complex. I have tested all of them, at least I think I have, which means that you might find omissions, mistakes, inaccuracies... If you happened to come across such failings, please send me an email at cppArduino@md-dsl.fr explaining what it is you think is wrong. I shall look into it, try to respond, and bring in corrections for the next edition of this book, currently 1st edition.

My Web site account contains the frameworks, and *Awk* and *Perl* programs presented in these two books. To download these, go to md-dsl.fr/c-arduino-programming (a little over 1.5 megabytes).

You will also find notes and acknowledgements as to events concerning these two books in my Web site md-dsl.fr.

This page intentionally left blank

About the author

The author, Michèle Delsol, born in France, educated in the USA (MIT - B.Sc., Sc.D.), now retired and living in France, has worked in industrial firms in South America, the USA, and France. For the last 25 years of her career, she was CEO and CTO of the company she created. As CTO she gained experience working with Fortran, Fort, Lisp, Java, JavaScript, Visual Basic, PHP, HTML, and C++.

Not shying from rolling up her sleeves, she is practical and dives into hands-on work. Her passion for flying led her to build and fly her own airplane (RV8). She also claims some artistic capabilities (the warthog and marmoset on the covers of this two-book tandem are hers), did some acting (theater), and is now an enthusiastic beekeeper.

Her most recent electronics endeavor is an *Arduino-based beehive weighing system* to help monitor her bees' wellbeing (full details in the author's book *Defensive C++ Arduino Programming* - see *Bibliography - C/C++ programming* (page 263). It began as a back of the envelop idea which grew into a full-fledged system. As the project progressed, from proof of concept of the individual components to a comprehensive integrated system, the application grew to more than 35 files (15,000 lines of code). Her early development was fairly undisciplined and incremental, which led her to too much time debugging - what was initially an enjoyable pastime became a gruesome burden. She consequently stepped back and researched why programmers make mistakes. She found that she needed to adopt good programming practices and use better tools. It brought her to switch to AtmelStudio (later to Visual Studio with Visual Micro) in lieu of the Arduino IDE, to extensively review C++, to create frameworks to handle specific tasks, to relearn adequate *Awk* and *Perl* to extract documentation from the source files, to learn regular expressions, and to undertake other useful programming chores.

This experience led her to write extensive notes on the material she covered to keep track of things as literature and the Internet teemed with too much material, most of which was unprofessional, verbose, and did not address the question posed directly. Ask a simple question, get long complicated responses. These notes gradually morphed into two books: *Pragmatic C++ Arduino Programming*, a reference work to help already C++ savvy Arduino programmers avoid the many gotchas C++ can throw at them, and *Defensive C++ Arduino Programming* which presents C++ tools and frameworks to improve programmer productivity to write efficient, robust, maintainable, and compact Arduino applications.

Her *Arduino-based beehive weighing system* is not quite finished. Future work would focus on transferring her project to an *Open Source Software team*, on redesigning the PCB for SMT technology, on implementing alternate communications to handle remote areas where GSM is not available, on developing Web and smartphone based user-friendly interfaces, and more. Whether these will be undertaken remains to be seen since she has other projects she intends to work on. Amongst the many ideas that float in her mind, a device to detect Asian wasps hovering in front of the beehives. She also plans to interpret a colony's activities via the sound they make. These projects are a tall order - they imply learning and implementing digital signal processing, digital image processing, and AI. These should keep her busy for years to come. Needless to say, she is never bored. Aside from the shared benefits that beekeepers might gain from her hard-won endeavors, her ongoing satisfaction lies in the challenge and the doing.

This page intentionally left blank

Index table

Symbols

++ See Prefix/postfix operators (++/--)

+ plus sign

=- instead of == or += instead of += - gotchas 221

+ plus symbol

+a - unary plus (affirm positive) 65

a+b - add 65

<, <=, >, >= operators

Less than, less than or equal with, greater than,
greater than or equal with 65

<< and >> bit shifts operators

Bit shifts left and bit shifts right 65, 70

< left chevron symbol

<<=, >>= cumulative bit shift left, bit shift right 65

= equal sign

+=, -=, *=, /=, %= operators - cumulative add/
subtract, multiply/divide, modulo 65

==, != operators - equal with, not equal with 65

=- instead of == or += instead of += - gotchas 221

> right chevron symbol

<<=, >>= cumulative bit shift left, bit shift right 65

~ tilde

~ bit NOT operator reverses bits 65, 70

! ~ negation, also used in bitwise not 65

3rd generation programming languages See Programming languages

255 values

Noise (255 values) in transmission filter out 284

Use 255 base instead of 8-bit 256 base 284

& ampersand

&, ^, | bit-level AND, XOR, OR 65

&=, ^=, |= bit-level logical self comparisons 65

&, |, ^, ~ bit-level AND, OR, XOR, NOT 70

&&, ||, ^ logical operators AND, OR 65

&a - address of variable a 113

& instead of && and | instead of || or vice 221

/*...*/ and //... See Comments

\ backslash symbol

\" double quotes escaped 52

\n, \t, \f (new line, tab, form) - escape sequences 52

Write macro on several physical lines 171

__brkval and __flp

System vars define fragmented memory 83, 164

(cast) operator

cast such as (uint8_t) myVal 65

: (colon uses)

>:0 - start at byte boundary - bitfields 94

:1 - pad 1 bit - bitfields 94

Bitfield size such as int myBitfield : 3; 69

Colon : used in 'for (int valItem : floatArray)' 75

? : - conditional if true/false 65

Conditional operator ? : contains colon : 69

Enumeration size specifier : uint8_t {...}; 69

for (index : array) {...} index is control parameter 75

Identifies a label statement 76

Inheritance derivedClass : public baseClass 69

Initializer list myClass() : val1(0), val2(10) 69

: used to define enum item type 97

.cpp files (code files) See Header files (.h) and code files (.cpp, .ino)

#define, #undef See Macros (#define)

:: (double colon uses)

namespace scope :: resolves name clashes 38

::new - global new 163

#error See Macros (#error)

! exclamation mark

==, != operators - equal with, not equal with 65

! ~ negation, also used in bitwise not 65

-f flags (compiler options)

-fno-exceptions flag in platform.txt 25

-fno-rtti and -frtti flags (variable type info) Arduino 147

unsupported 224

-fpermissive required by Arduino 25, 202, 234

__FILE__

Current source file 83, 182

Full path consumes too much memory 191

__FILE__, __FUNCTION__, __LINE__

System variables 83

__flp and __brkval

System vars define fragmented memory 83, 164

/ forward slash

+=, -=, *=, /=, %= operators - cumulative add/
subtract, multiply/divide, modulo 65

/ forward slash symbol

*, /, % operators - multiply, divide, modulo 65

-fpermissive flag (compiler leniency) See Permissive flag (compiler leniency)

__func__ (deprecated) See __FUNCTION__

22 | Index table

FUNCTION			
Compatible with F() macro	188	Array index [] operator overloading	146
Current function	83, 182	Array [] type qualifier - int myArray	57, 94
.hex (image) See Executable file		[](){...} - lambda function operator	65
.h files (header files) See Header files (.h) and code files (.cpp, .ino)		throw(error) in overloaded index[] operator	148
#ifdef...#elif...#else...#endif See Macros (#ifdef...#elif...#else...#endif)		* star symbol	
#if defined (...) See Macros (#if defined (...))		* defines a pointer	56
#ifndef...#endif See Macros (#ifndef...#endif)		*myPointer++ - is it (*myPointer)++ or *(myPointer++)?	225
#include See Macros (#include header file)		*, /, % operators - multiply, divide, modulo	65
.ino files (sketches)		* symbol	
Arduino name for C++ entry file	103	+=, -=, *=, /=, %= operators - cumulative add/subtract, multiply/divide, modulo	65
Header files (.h) and code files (.cpp, .ino)	103	% symbol	
.ino files are ordinary C++ files	40	+=, -=, *=, /=, %= operators - cumulative add/subtract, multiply/divide, modulo	65
LINE		#undef See Macros (#define, #undef)	
Current line no. in source file	182	A	
Current source line	182	abort	
#, ##, /**/ macro operators		Error handling	145
Macro stringizing, concatenation, separation	177	Abstract classes See Virtual & pure virtual functions	
- minus sign		Address '&' operator	
+=, -=, *=, /=, %= operators - cumulative add/subtract, multiply/divide, modulo	65	Address of as in '&a'	113
-= instead of = or += instead of += - gotchas	221	Advanced mechanisms	
- minus symbol		Complex numbers	150
a - b - subtract	65	Exception handling (C and C++)	77, 147
-a - unary minus (affirm negative)	65	Lambda functions [](){...}	65
, operator		Templates	125
Sequence expressions (comma operator)	65	Aliases (uint8_t, etc.)	
? : operator		Types (built-in)	54
Conditional if true/false operator	65	Amateurism	
*, & operators		Doing a job without the required skills	209
Dereference a pointer or define one, address of a variable	65	Too much debugging time	257
& operator See Address '&' operator, reference type qualifier, bit level AND operator		Analog pins See Arduino specific functions	
{...} operator See Curly braces		AND & and && See bit-level and logical operators	
() parentheses		Arduino bit functions (simplify programming)	
() parentheses defines function - int myFunc()	57	bitClear, bitSet, bitRead, bitWrite, bit(pos)	89
() parentheses groups expressions	65	bitRead(number, bit) to get a bit's value	71
% percentage symbol		Divide by 256 for MSB or modulo 256 for LSB	89
*, /, % operators - multiply, divide, modulo	65	Arduino Builder See Code::Blocks	
#pragma See Macros (#pragma)		Arduino configuration file See platform.txt	
& reference qualifier See Reference qualifier		Arduino development tools See Development tools	
-- See Prefix/postfix operators (++/--)		Arduino editor	
[] square brackets		Adds 'main' behind the scenes	40, 44
Array [] type qualifier - int myArray[50]	56	Arduino editor collapses curly braces well	106
Overloaded index [] operator SafeArray class	276	Auto-indenting, code collapsing not supported	22
[] square brackets symbol		Error message lost in reams of output	200, 254
Array index operator - myArray[2]	94	Good C++ editor but could be better	21, 197
		#ifdef auto-indent/code collapse unsupported	218
		See Arduino IDE bugs	197
		Arduino IDE	

All-in-one free user-friendly tool	5, 19	Arduino bit functions	89
Arduino boards based on Atmel 8-bit chips	11	Arduino port functions to manage ports	84
Arduino IDE's messages are not always clear	249	Digital and analog I/O functions	83
Bit-level coding	71	Digital vs analog ports	11
Complex numbers not supported by Arduino	151	Interrupts	83
Experiment with arrays to find code size	282	Microcontroller specific functions	83
Preprocessor, powerful text processing tool	20	millis(), micros(), delay(), timers	83, 85
See Arduino IDE tool chain	19	Nomenclature: port for boards, pin for chip	84
Warnings on	224	Port configuration digital/analog input/output	84
Arduino IDE alternatives		PWM ports	84
AtmelStudio, Visual Studio, Visual Micro, VS code,		Random numbers	83
PlatformIO, Code::Blocks	32	Serial and stream communications	83
Arduino IDE bugs		Signal functions	83
Adds unwanted curly brace	220, 244	Some boards can put out analog voltages, not	
Arduino's segmentation fault (linker problem)		UNO nor MEGA2560	84
pushed me to migrate to AtmelStudio	197, 199	Use Aref port as voltage reference in analog read/	
Error messages and warnings	198	write	84
Error reporting top down, errors out of sight	254	Voltage by PWM pulse width modulation	84
if/else open curly brace position problem	22	Arithmetic on array items See Ordinary arithmetic	
No alert on externally modified file	198	vs. pointer arithmetic	
Sketch's serial port gotcha	198, 203, 204	Arithmetic operators	
Stray '\357' injected by AtmelStudio	197	+, -, *, /, % - add, subtract, multiply, divide, modulo	65
Stray '\' in macro definition	197	Arrays and indices	
Suddenly stops working (java error)	200	Array of pointers	82
Undo - ctrl-Z sabotages your work	197, 198	Array qualifier [] differs from pointer *	57
Upload error need recompile	197, 200	Arrays auto-sized by initializations	102, 123
Will not alert you on externally modified file	204	Arrays group identical items	151
Arduino IDE (build)		Arrays group identically typed items	94
Compile/upload uses bootloader to upload		Array type qualifier [], index operator []	66, 82, 94
program	27	Associative containers not supported	151
Arduino IDE tool chain		C++ building-blocks	50
Arduino C++ editor	20	Memory corruption due to runaway index	245
avrdude (uploader)	21	Nested array initializations	95, 124
AVRlibC Atmel 8-bit specific library	99	Number of items in array	70, 82
bootloader & program stored in flash memory	21	One- and three-dimensional arrays	95
Breadboard to prototype board to final PCB	5	Overloaded index operator[]	101, 129, 146
Compiler	20	SafeArray class protects from bad index	238
Debugging (hardware-based)	21, 29	Size of array passed as function parameter	220
GNU C++ tool chain and avrdude	99	sizeof operator used on array	70
Linker - creates program by assembling .o files	20	Strings in array vs. corresponding enum list	235
make utility (builder utility)	21, 26	Type qualifier [] converts variable into an array	79
Proof of concept - work on breadboards	5	Use subscript (index) to get value of item	83, 95
Python and WiPy	16	Zero-based indexing forgotten	220
Serial terminal	21, 29	Artificial intelligence See ChatGPT	
Several tools behind Arduino IDE interface	19	Assembler See Programming languages	
Arduino memory pools		Assignment = as initializer See Initialization	
Bottom of RAM for global, static, system data	154	Associative containers See Arrays and indices	
PROGMEM for read-only variables	98, 153, 154	Associativity See Precedence and associativity	
RAM is volatile, flash and EEPROM remain	154	ATmega328P-Xmini, ATmega256RFR2, Atmel ICE	
Sizes of memory pools table	154	Debugging (hardware-based)	29
Three memory pools: RAM, EEPROM, flash	154	ATmega microcontroller interrupts	
Arduino specific functions			
Analog read and analog write	84		

24 | Index table

Atmega328P has 2, Atmega2560 has 6	86	AVRLibC See Libraries	
AtmelStudio		B	
Advanced code management features	32	bad_alloc, bad_cast	
Alerts on externally modified file	204	Exception class events	149
Arduino IDE interoperability	5, 32	Base class See class (base)	
Arduino's segmentation fault pushed me to migrate to AtmelStudio	199	Basic types See Types (basic)	
AtmelStudio has a very good editor	32, 282	Beehive weighing system	
AtmelStudio segmentation faults are benign	200	Or how I learned from my mistakes	207
Code navigation features	32	Preface	xix
Collapses and indents #ifdef, not Arduino	216	Binary decision trees	
Color syntax	22	if (condition) {...}	74
Compiler options	25	Binary numbers See Integers, floats, etc.	
Defensive C++ Arduino Programming	32	bitClear, bitSet, bitRead, bitWrite, bit(pos)	
Experiment with arrays to find code size	282	Arduino bit functions	89
Free download from Microchip Web site	32	Bitfields	
GNU C++ tool chain and avrdude	32, 99	:0; start at byte boundary and :1; pad 1 bit	94
Importing Arduino projects is hassle free	5, 32	Byte and word boundary	94
Indents #ifdef, not Arduino	218	class and struct	91
Manage upload process	27	enums simplify access to bitfields	93
Microsoft style environment	32	Pack small numbers in bitfields	93
Serial terminal	29	Bit-level coding	
Stray "\357" injected by AtmelStudio	201	Arduino bit functions simplify programming	71
Supports Arduino hardware-based debug	29, 32	Bit operators	67
Supports name completion	22	Bit shift precedence ignored bad result	220, 241
User-friendly error reporting	32	Date and time packed data	132
AUTO_CHAR_ARRAY		GET_BIT_VALUE_POS(data, shift)	71
Adds extra null byte to terminate string	183	Bit-level operators	
Use macro to not forget end null	240	<< and >> bit shifts operators	70
Zero-based indexing forgotten	240	Arduino bit functions	89
Automatic memory allocations See Functions and variables		Bit-comparison, inversion, and shift (operators)	70
auto type qualifier		Manipulate individual bits in a byte	70
Sets variable's type automatically	55, 79	Read/set bits via AND, XOR, OR (&, ^,)	65, 70
Type checking	219	Store/set bit values - three methods	242
Use auto to set type and initialize	58	Bit masks	
avrdude		Bitfield date/time access	133
Arduino compile/upload uses avrdude to send .hex file to board, bootloader uploads it	27	Read/set bits via masks	70
Arduino Uno used as ISP via bootloader	27, 28	Bit notation	
bootloader, located at address 0, transfers control to program or uploads new program	27, 28	8-bit representation of 1 is B00000001 (right to left)	242
GNU C++ compiler/linker create .hex file	27	Bit display is zero-based	71
Program resides in flash memory (.hex file)	27	Serial.print BIN (binary) print format	88
Reset sends first program instruction to register for program execution	28	Bjarne Stroustrup	
Uploading a program requires an in-system programmer (ISP); bootloader does the job	27	Operator associativity not defined	68
avrdude (uploader)		PhD thesis	8
Arduino IDE (tool chain)	21	The C++ Programming Language book	150
bootloader resides in flash memory, uploads program into the microcontroller	27, 185	Where the ++ in C++ comes from	9
Upload error need recompile	201	Blynk for Arduino See Visual development	
		Book's Web site (md-dsl.fr)	
		Events concerning book blogged in md-dsl.fr	291
		For comments and info, please send email to cppArduino@md-dsl.fr	291

Go to https://md-dsl.fr/c-arduino-programming for a link to download most of the code in the two books	291	C functions must have different names	120
bootloader See avrdude		Creating structs in C and C++ differ	39
Boot See Powerup		Enumerations (enum)	39
Bottom of available RAM See Memory (heap)		extern "C" to specify legacy C code	46
Bottom of physical RAM See Arduino memory pools		extern "C" {...} triggers explicit C compilation	39
Bottom of stack See Stack and stack frames		malloc is a function, new is a C++ operator	39
Bottom-up See Top-down programming		Name mangling differentiates functions	38, 120
Breadboard See Arduino IDE		namespace	38
Builder utility See Arduino IDE tool chain's make utility		C++ features not supported by Arduino	
Byte boundary See Bitfields		Associative containers	151
C		Exception handling (C++)	150
C++ building-blocks		I/O streams	151
Arrays and indices	50	Multitasking and concurrency	151
C++ arrays are zero based	41, 82	Regular expressions (regex)	151
class and struct	50	Standard Library and Standard C Library	150
Code-blocks defined inside curly braces	72	C++ mechanics	
Comments /*...*/ and //...	49	Array initialization	102
Components used to build a program	49	C++ arrays are zero based	41, 82
Control flow statements	50	C++ has strong typing	41
.cpp files (code files)	101	C++ is case sensitive	41
Create derived types via type qualifiers	49	C vs. C++	44
Data packing (bit-level)	50	Exception handling (C and C++)	44
Derived type from type qualifier on basic type	56	Functions and variables	102
Enumerations (enum)	50	Functions do not necessarily return a value	42
Expression evaluation	49	Manipulate data addresses directly	41, 81
Functions and variables	50	Name mangling differentiates functions	120
Header files (.h) and code files (.cpp, .ino)	101	Naming functions, variables, macros	110
.ino files (sketches)	101	Numeric types (compatible)	102
Lambda functions can simplify programming	143	Object-oriented programming (top-down)	15, 40
Libraries contain most of what you'll need	51	Operator overloading	101, 129
Operators (logical and bit-level)	49	Ordinary arithmetic vs. pointer arithmetic	102
PROGMEM framework	98	Param passing: value, address, reference	42, 102
Statements as collection of expressions	49	Polymorphism	102
Structure declarations not nested as in C	45	Precedence and associativity	101
Templates - function/classe blueprint	102	Programs are multi-module	40, 41, 101
Types - built-in and user defined	49	Put declarations anywhere before use	39
Unions	50	Rules to assemble C++'s components	40, 101
Use PROGMEM for read-only variables	50	Scope (visibility)	101
What one needs to master	2	Strings	42, 102
C++ core guide lines		Type checking	102
Do's and don'ts in C++	xix	Types (user-defined)	43
C++ enhancements to C		What one needs to master	2
bool is C++	39	C++ programming	
C++ compiler triggers errors on C programs	38	Akin to walking through a minefield	209
C++ enables creating user-defined types	17	Bjarne Stroustrup's The C++ Programming Language book	10
C++ is an enhanced C yet there are differences	38	C++ is an enhanced C	38
C and C++ programming almost identical	44	Compromise between programming ease and execution speed and compactness	9, 12
C declarations in top of a file or function	39	C with user-defined types	38
		Programming languages	12
		Top-down programming	17
		Which chip/language combination?	14

26 | Index table

C++ short history		Virtual & pure virtual functions	136
Bjarne Stroustrup where ++ in C++ comes from	9	class (templates) See Templates (class)	
C's origins (multics then unix) make it unique	7	Code-blocks	
C with classes (C++) for top-down programming	8	Classes, functions, structures, program control	
Good code needs to be simple	9	flow, etc. - program units do work (statements	
How one thinks (top-down)	8	enclosed in curly braces)	72
Inheritance	9	Nameless code-blocks improves readability, saves	
Multitasking and concurrency	9	RAM	72
User-defined types basis for object-oriented programming	8	Code::Blocks IDE	
C and C++ numeric types		Arduino Builder (freematics): compile/run Arduino projects	34
Table	54	Seems promising, but unable to use it as alternative for Arduino development	34
Cascading values		Code files See Header files (.h) and code files (.cpp, .ino)	
Data sequence and #define	174	Code folding See Curly braces	
catch See Exception handling (C and C++)		Cognitive dissonance and EGO	
C# (C-sharp)		Ex: bitfields error psychologically induced	270
Microsoft's C++ equivalent	10	One can be one's worst enemy	260
char constants See Constants: character, string, and numeric		Psychological factors	258, 271
char string and PROGMEM See Strings		Collapse items See Curly braces	
char type See Types (basic and derived types)		Colon uses See beginning index table	
ChatGPT		Color syntax	
Alternative coding solutions with ChatGPT	268	Arduino editor and AtmelStudio	22
Artificial intelligence search tool	36	Comments	
ChatGPT writes temp/humidity program	36	C and C++ comments (/*...*/ and //)	49, 51
Generates code from simple design requests	36	Careful with C++ comments '//' in macros	171
Kick start you application by asking ChatGPT	268	Insert comments when writing code	51
One more tool to help you write better code	268	Communications See Serial Communications	
C language		Compatible numeric types See Types (equivalent numeric types)	
Procedural programming	16	Compiler	
class and struct		Arduino IDE (tool chain)	20
Bitfields	91	Converts .cpp to machine code .o	25
C++ building-blocks	50	Optimizes generated machine code	26
class and struct almost identical, use struct for light weight data aggregates	91	#pragma macro sets compiler directives	182
class and struct create user-defined types	56, 91	Semantics (vocabulary) and grammar (rules)	25
class array initialization	102	Compiler bug?	
const functions can modify mutable variables	61	Cannot where curly braces inbalance	245
Constructors and destructors	91	Compiler points far from error's position	218
Data encapsulation key to top-down programming	38, 89, 123	float to uint32_t problem using pow()	238
Inheritance	91, 134	Misleading message from enum declaration	255
private/public control class items access	91, 135	No code after label error	236, 251
class (base and derived)		No error message on missing return	111, 235
Base class stores common properties, derived class stores specific properties	136	No warning on redefining existing variable	227
Cycle through derived classes	137	No warnings on mixed compatible types	121
Derived class constructor initializes base class	135	Omitting () in function call undetected	220, 232
Derived class redefines base class virtual functions	120, 134, 136	Problem not detected because error is legal	218
Inheritance defines parent/child relationship	134	Unable to find a workaround for the Arduino segmentation fault linker problem	199
Transparently cycle derived classes	138	Compiler bug See Error messages and warnings	

Compiler optimizations		WolfPack template class example	127
Circumvents weaknesses in your code	225	Containers See Arrays and indices	
Compiler optimizes your code	225	Contiguous heap See Memory (heap contiguous)	
volatile type qualifier no optimizations	63	Control flow expressions	
Compiler options		C++ building-blocks	50
Debugging and release modes	25	Control flow constructs handle program logic	73
-ftlo flag (link time optimization)	25	do {...} while (condition)	74
-fno-rtti flag to activate the typeid operator	224	Exception handling (C - setjmp/longjmp)	74
-fpermissive flag (compiler leniency)	25	for (each) {...}	74
Optimization level	25	for (iterate) {...}	73
Tweak the compiler to meet your needs	25	goto label	74
Verbose produces extensive warnings	25	if (condition) {...}	73
Complex numbers		Scope (visibility)	106
Arduino missing complex numbers class	150	Statements (if, while, etc.)	72
Get from GitHub or create your own complex numbers class (easy)	150	switch (value) {...}	74
Operator overload to add complex numbers	130	try (...) exception handling (C++)	73
Concatenation operator ##		What defines C++	18
Macro concatenates items	178	while (condition) {...}	74
Concurrency See Multitasking and concurrency		C programming	
Conditional if true/false operator		3rd generation languages	10
? : operator	65	Cryptic messages See Error messages and warnings	
Conditional inclusions See Macros (#ifdef...#endif)		ctrl-Z (undo) bug See Arduino IDE bugs	
const and mutable		cumulative bit shifts operators	
const before/after pointer * modifier	61, 62	<<=, >>= cumulative bit shift left, bit shift right	65
const_cast to modify variable declared const	61	Curly braces	
const functions and variables protect data	61, 63	Arduino editor adds unwanted curly braces	244
Functions and variables	61, 81	Collapse code improve code readability	106
mutable prevents/allows makes data modifiable	63	Comment closing curly brace	243
register type qualifier (deprecated) applied to variables which impact speed	63, 81	Compiler cannot find error location	243, 245
volatile type qualifier no optimizations	63, 81	Curly brace instead of semicolon not done	245
Constants (character, string, numeric)		Curly braces {...} as initializer	79
C++ provides three types of constants	49	Curly braces define local scope	104
char constant is 16 bits in C, 8 bits in C++	46, 52	Encapsulate function body in curly braces	110
Decimal value	52	Readability is enhanced via curly braces	106
Escape sequences	52	Serial does not name type bad curly braces	244
Global, static, system data bottom RAM	155	Unbalanced curly braces	244
Literal string constants	52	C vs. C++ See Syntax differences between C and C++	
Numbers and literal string constants	52	Cycling through derived classes See Virtual & pure virtual functions, abstract classes	
Constructors		D	
class and struct	91	Data corruption See Memory corruption	
Constructor initializes; destructor wraps up	39, 79, 92, 123	Data packing	
Constructors create instances of structs and classes, may take parameters, no return	92	Bit level - Use individual bits to pack data	50, 129
Constructor's name same as class's name	92	Date and time packed data table	133
Copy constructors penalize runtime size	118	Group date/time inside a few bytes	132, 133
Member initialization list	135	Group small numbers in bytes via bitfields	131
See destructors	92	Table of data packing into bitfields	132
See SafeArray class example	277	Data storage	
		Data pointer's value should not be zero	227
		Memory allocations must be monitored	157

28 | Index table

unions enable memory sharing	96	Delphi See Programming languages	
Ways to store data: automatic (function variable), global, static, heap, stack, PROGMEM	156	Dereferencing a pointer	
Debugging (general)		Dereferencing a pointer has low precedence	219
Amateurism causes extra debugging work	257	Dereferencing * function argument bad	226
Defensive C++ Arduino Programming	29	*myPointer++ Is it (*myPointer)++ or *(myPointer++)?	225
Devil is in the details	229	Parentheses ensure order of evaluation	225
Why does one make mistakes?	208	Pass by address	114
Debugging (hardware-based)		Pointer arithmetic, careful with precedence	225
Arduino IDE 1.8.19 does not support hardware- based debugging, V 2.0 does	21, 29	Pointer wrongly passed as parameter	219
ATmega328P-Xmini, ATmega256RF, Atmel ICE	29	Derived class See class (derived)	
AtmelStudio supports hardware-based debugging	29, 32	Derived types See Types (derived)	
PlatformIO does not support Arduino hardware- based debug	34	Destructors	
PlatformIO supports ESP32 hardware-based debugging, not Arduino	29	Does wrap-up chores, releases memory	39
Debugging (print-based)		No parameters and no return value	92
Hardware- and print-based debugging comple- mentary	29	Object's destructor does wrap-up chores	92
Relies on extensive use of macros	170	See constructors	92
Selective code inclusion via #ifdef macros	210	Development tools	
Declarations		Arduino IDE	19
C declarations - top of a file or function, C++ anywhere before being used	39	Artificial intelligence (ChatGPT)	35
Declaration examples	79	AtmelStudio	32
Identifier (variable must have a name)	78	Code::Blocks	34
Tells the compiler how to call a function and implement it using its stack frame	78	MPLAB	34
Typedef synonym for complex declaration	78	PlatformIO	33
Type qualifiers in multivariable declarations must be repeated	223	Visual development	35
Types (derived and user-defined)	78	Visual Micro for AtmelStudio (MicrochipStudio)	32
void functions	79	Visual Micro for Visual Studio	33
Declarations vs. implementations See Header files (.h) vs. code files (.cpp, .ino)		Visual Studio	33
Decrement/increment --/++ See Prefix/postfix op- erators		VS Code (Visual Studio Code)	33
Default function return type See Functions and variables		Development tools See Artificial Intelligence: ChatGPT	
Defensive C++ Arduino Programming book		Development tools See Visual development: Visuali- no, Scratch, Blynk	
AtmelStudio	32	Devices, sensors, hardware support	
Debugging (hardware-based)	29	Which chip/language combination?	15
Exception handling (C and C++)	78	Digital and analog I/O functions See Arduino specif- ic functions	
Frameworks	210	Digital pins See Arduino specific functions	
Perl and Awk extensively covered	47	DOS box	
Pseudo-exception handling framework	280	Awk, Perl, grep, sed, avrdude	204
Regular expressions (regex) covered	151	Command line apps need text interface	204
Tools and procedures to optimize programming	1	Windows command processor	204
delay() See Timers		Double colon uses See :: (double colon uses) - be- ginning index table	
delete See new and malloc		do {...} while (condition)	
		Control flow statements	74
		Iterates at least once	76
		E	
		EEPROM	
		eeprom_write_block and eeprom_read_ block -store/retrieve data from EEPROM	168

Flash memory	185	Missing backslash: no error, no warning	213
Non-volatile memory	154	Missing closing far from error's location	243
Programmer can use EEPROM to store read-write variables permanently after power-off	154, 168	No code after label error	220, 236
EGO See Cognitive dissonance and EGO		No error message on missing return	111
Encapsulate data and functions		No warning on redefining variable	227
class and struct	38, 91	Stray '\357' injected by AtmelStudio	201
End null See Strings		Upload error need recompile	201
Enumerations (enum)		Escape sequences	
C++ building-blocks	50	Constants - character, string, and numeric	52
class keyword used as enum scope qualifier	97	\" double quotes escaped	52
Define low/high enum list boundaries	97	\n, \t, \f (new line, tab, form feed)	52
enum EColor example	97	ESP32	
Instead of #define sequence, use enums	97	PlatformIO	34
List of symbolic constants each with a value	97	Which chip/language combination?	14
Simplify access to bitfields	93	Event registration	
static_cast assigns arithmetic value to enum	98	Error handling	145
Symbolic numeric constants interchangeable	97	Exception class events	
: used as enum item type	97	bad_alloc, bad_cast, out_of_range, range_error, overflow_error, underflow_error	149
Use symbolic enums to index into arrays	95	Exception handling (C and C++)	
Equivalent numeric types See Types (equivalent numeric types)		C++ exception unsupported by Arduino	147, 150
Error handling		Control flow statements	74
abort	145	Defensive C++ Arduino Programming	78, 280
Cascade back to some predefined location (C and C++ exception handling)	145, 147	Do extensive error checking	147
Decide what to do next	145	Error context defined by jmpbuf	78
Dedicated error handling function	146	Error handling	145, 146, 147
Developing logic differs from handling errors	145	Example code	77
Good programming practices	47	Exception class	149
goto label	76	longjmp triggers roll back to setjmp	77, 147
Issue warning, continue with default	145	noexcept specifies no exception thrown	149
Never assume anything	129, 145	Replaces C++ exception handling	73
Provide info on the nature of an event	145	SafeArray class does try/longjmp	146, 279
Use setjmp/longjmp (exception handling)	145	throw added as C++ exception handling	146
Error messages and warnings		throw called by overloaded array [] operator	148
Arduino IDE bugs	198	throw (errorID) gets caught by catch	146, 147
Bad enum declaration error message	255	throw operator in C++ exception handling	65
Cryptic message following enum error	255	throw's errorID identifies individual throws	147
Error messages can be long and cryptic	249	throw used in SafeArray class	279
Error reporting top down, errors out of sight	254	try/throw/catch	73
Expected ';' before '{' - initializer left in	249	Use setjmp/longjmp instead of C++ exception handling	149, 279
Expected ';' before '{' - initializer left in	252	Executable file (aka program)	
Expected initializer before 'xyz' no ';'	249	bootloader resides in flash memory, uploads program into the microcontroller	27
Expected primary expression before 'char Foo(10, char* msg);'	250	Linker assembles .o files into program	26
Expected primary expression before '}' - label gotcha	251	Program stored in flash memory	185
Expected unqualified-id before '{' token - missing first '\ ' in macro definition	250	Expression evaluation	
Inaccessible member base class not public	255	C++ building-blocks	49
Invalid char(*)[4] to uint16_t	201	Gotchas (C++)	67, 101
Macro undefined - no parameters when called	251	Overflow/underflow in expression evaluation	67
		Precedence and associativity	101
		extern "C" {...} See C++ enhancements to C	

F		Functions operate on data	81, 109
Flash memory		Function stack frame at bottom of stack	111
Non-volatile memory	154	Function stack frame bottom of stack	155, 167
RAM use with & without the F() macro	189	Lambda functions <code>[](){...}</code>	80
Save RAM with PROGMEM to store read-only data in flash memory	98	mutable allows functions to modify variable	81
Sketch uses 'nnnn' bytes	157	Name mangling differentiates functions	38, 120
Used to store bootloader, program (executable file), and PROGMEM variables	154, 185	namespace: use variables and functions with same name from different libraries	144
Flash, RAM, and EEPROM sizes		Param passing: default value	110
Table memory pool sizes	154	Param passing: value, address, reference	112, 115
Float numbers See ints, floats, octal, hexadecimal, binary		Pass a pointer and dereference	111
F() macro		Passing array to function requires passing array size for index validity checks	220, 237
FILE and FUNCTION	188	Recursive functions	165
Macro's mem requirements as program runs	188	register type qualifier (deprecated) applied to variables which impact speed	81
PSTR() macro for read-only variables	189	Runaway index damages system, heap, stack	160
Save RAM on Serial.prints PROGMEM	155, 188, 191	Scope (visibility)	106
for (each) {...}		Specify return type otherwise function returns int no error message	111, 241
Control flow statements	74	static qualifier	80
Cycle through array, iteration is automatic	75	strcpy, strcat, strlen, strcmp functions	42
for (iterate) {...}		Typedef FPtrVoid fpHelloWorld	141
Control flow statements	74	Typedef is synonym for complex declaration	80
for to while transformation	75	Validity checks on parameters passed	111
Iteration controlled by an index	73	volatile type qualifier no optimizations	81
Fragmented heap See Memory (heap fragmented)		Function templates See Templates (function)	
Frameworks See Book's Web site to download		G	
FreeList		Generated code	
List of fragmented memory holes	165	avrdude uploads app into microcontroller	21
Freematics.com See Code::Blocks		Compiler optimizes your code	26
free See new and malloc		Linker optimizes away unused items	192
Function ()		GET_BIT_VALUE_POS(_data, _shift)	
Type qualifier transforms variable into function	79	Bit-level coding macro	71
Function pointers		GitHub	
Function pointer naming convention	140	Count for Arduino, RaspberryPi, ESP32	16
Typedef	140	Glitches See Gotchas	
When to create function pointers	141	Global and local scope	
Functions and variables		Global, static, system data in bottom of RAM	155
Auto allocation variables in stack frame	82, 155	Statements not allowed in global scope	253
C++ building-blocks	50	Store application wide data in global variable or via allocations to a global array pointer	156
class and struct	91	GNU C++ tool chain and avrdude	
Compiler removes intermediate variables	42	Arduino IDE (tool chain)	99
const functions and variables	61, 81	AtmelStudio	32, 99
Contiguous heap required for stack growth	227	GNU compiler uses AVRlibC Atmel library	99
Default function parameters	110	Golden rules	
Default function return type is an int	241	Always monitor memory	48, 153, 158, 228
Dereferenced pointer parameter wrong	112	Beware of cognitive dissonance (EGO trap)	47
Function calls via function pointers array	140	Check, Check, Check!	47
Function copies parameter in pass by value	111	Do not reinvent the wheel	48
Function () operator used to hold parameters	142	Encapsulate related data and functions	60
Function parameter int[] becomes int*	237		
Function pointer example: PrintSurface	142		

Exploit C++ features sparingly	47	Expected ';' before '{' - initializer left in	249, 252
In-line documentation is a necessity	47	Expected initializer before 'xyz' no ';'	249
Insert comments when writing code	51	Expected primary expression before 'char Foo(10, char* msg);'	250
KISS principle: or why complicate things	48	Expected primary expression before '}' - label	
Know yourself - listen to your body	262	gotcha	220, 236, 251
Never assume anything	47, 129	First and last items of an array zero-based	220
Never end a function without a return!	111	float to uint32_t problem using pow()	220, 238
switch should have coded default	76	Forgot semicolon	249
Think!	47, 267	Function masks variable in outer scope	219, 226
Update your C++ skills (know your tools)	47, 48	Glitches - often caused by bad pointers	219, 227
Good and bad habits		In += unary + interpreted instead of +=	212
Bad habits: easy - good habits: hard	261	Inaccessible member base class not public	134, 255
Introspection and self-imposed questioning	261	Incompatibility between function's defined and	
Psychological factors	258	called parameters	126
Good programming practices		Initializer {0} left in leads to cryptic message	252
Auto-indent systematically and check	209	Invalid char(*)[4] to uint16_t	197, 201
Comment closing curly brace	243	Lack of a terminating null thrashes memory	240
Comment closing #endif with #ifdef	218	Legal code yet programming error	207
Decide on error handling methodology	47	Legal code yet programming error	218
Do not neglect program documentation	47	*myPointer++ Is it (*myPointer)++ or	
Encapsulate code in curly braces	106	*(myPointer++)?	225
Function pointer naming convention	140	Omitting () in function call undetected	219, 232, 243
Functions should contain explicit returns	235	One-line multivariable declarations: missing repeat	
Is your mental condition up to par	209	type qualifier	219, 223, 254
Macros improve readability and reduce errors	210	Parameter declaration char* msg left in when	
Make names meaningful	25, 110	calling function	250
Modularize the application	210	Passing an array to a function requires passing	
Prefix parameters with an underscore	79	explicit array size for index validity checks	220
Professionalize your work	46	Pointer not set to zero after free/delete	229
Standardize code (names, formats, etc.)	46, 209	Pointer used without assigning memory space	
Use FREE and DELETE macros to release memory		(phantom object)	228, 229
and set pointer to zero	246	Pointer validity not checked	221, 227
Verify that allocated pointers are nonzero	246	Precedence and bit-level coding	220
Go programming language		Prefix/postfix operators ++/-- misunderstood	221
Supports concurrency (tasks run in parallel)	13	Return type does not match assignment	223
Gotchas (C++)		Runaway index corrupts memory	245
Arduino IDE adds unwanted curly brace	244	Semicolon not replaced with curly braces negligent	
Auto-indent reveals unbalanced curly braces or		copy/paste	220, 245
parentheses	220, 243, 244	Serial does not name a type error	244, 252
auto typing gets you to lose track of variable's		Signed/unsigned in expression do not match	223
type	224	sizeof operator on array misunderstood	70, 219
Avoid mixing numeric types	221, 223	Specify function's return type otherwise function	
Bad dereferencing function parameter	219, 226	returns int no error message	235, 236
Bad enum declaration error message	255	Specify return type otherwise function returns	
C++ is a deceptively simple language	207, 218	int no error message	79, 111, 220
C++ traps and pitfalls: classic errors	218	Stack thrashes top of allocated memory	159, 227
Call overloaded 'myFunction' is ambiguous	253	Stray '\357' injected by AtmelStudio	201
char strings in array concatenated by mistake	220	Symptom when damaged section used	202
class's closing curly brace missing	255	Trace indentations backwards to find curly braces	
Comma instead of semicolon bad	219, 231	bug	243
Data sizes in expression do not match	67, 101, 219, 224	Type checking leniency induced bugs	219
Dereferencing a pointer has low precedence	219		

32 | Index table

Why does one make mistakes	207	RAM but not program	153
Zero-based indexing forgotten	220	Header files (.h) and code files (.cpp, .ino)	
Gotchas (C++) See Error messages and warnings		C++ mechanics	101
Gotchas (C++ traps and pitfalls)		class and struct declarations	101
-= instead of = or += instead of +=	221	Compiler works on one source file at a time hence	
a = b instead of a == b or vice versa	221	the need for declarations (header files)	103
"a" vs. 'a' double quotes instead of single	222	Declarations enable the compiler to verify correct	
Do not to start number with 0 unless is octal		parameter passing	103, 104
number such as 013 (decimal 11)	221	Ensure header files top-down dependencies,	
if...else badly constructed (dangling else)	222	avoid interdependencies	104
& instead of &&, instead of or vice versa	221	#ifndef avoids repeat file inclusions	104, 176
Missing end of statement or one too many		.ino files are ordinary C++ files	40
semicolons	222	.ino files (sketches - Arduino's C++ entry file)	103
Gotchas (macros)		Macros (#include header file)	24, 176
AtmelStudio color syntax detects macro bug	181	Program components declared in header files (.h),	
C++ comments in macros misleading	213	defined in code files (.cpp, .ino)	101, 103, 104
Commenting out part of a macro	210, 213	Heap (contiguous) See Memory (heap contiguous)	
#define tokenization creates extra spaces	210, 212	Heap (fragmented) See Memory (heap fragmented)	
#error gets you to look for macro definition		Heap See Memory allocations	
problem whereas the macro is fine	181	Hello World	
Expected unqualified-id before '{' token - missing		If you can create and run a small program, you can	
first '\ ' in macro definition	250	do the same with a big one, see Kernighan and	
Forgot backslash in multiline macro	210, 213, 252	Ritchie	xix
Functions called in macro generate side effects		Smallest C program	39
	210, 214	Hexadecimal numbers See integers, floats, octal,	
Macro definition errors often undetected	210	hexadecimal, binary	
Macro's simplicity can cause subtle errors	210	Holes (memory) See Memory (heap fragmented)	
Macro undefined - no parameters when called	251	I	
One too many backslashes in macro	210, 253	if (condition) {...}	
Operator precedence problem in macro	215	Binary decision trees	74
Semicolon in macro gotcha undetected	212	Control flow statements	74
Stray '#' in program	212	Image (.hex file) See Executable file (aka program)	
Tokenization in #define creates extra spaces	212	Implementations vs. declarations See Header files	
Unbalanced #ifdef...#endif pairs	211	(.h) vs. code files (.cpp, .ino)	
Unsatisfactory macro parameters isolation	211	Incremental vs. planned programming	
Gotchas (macros) See Error messages and warnings		Easy to type code without prior thinking	267
goto label		Incremental learning	268
Control flow statements	74	The most difficult thing to do is think!	267
Direct program transfers - a no-no	76	Increment/decrement ++/-- See Prefix/postfix oper-	
Error handling	76	ators	
Expected primary expression before '}' - label		Index operator overloading See SafeArray class	
gotcha	251	Index See Arrays and indices	
goto statement	73	Inheritance	
Spaghetti code procedural coding nightmare	76	Base class initialization	135
Grammar (rules)		Base class stores common properties, derived class	
Compiler	25	stores specific properties	135
Semantics (vocabulary)	25	class and struct	91, 134
H		Colon ':' in class definition defines inheritance	135
Hardware-based debugging See Debugging		Defines parent/child relationship	134
Harvard architectures		Traverse array of distinct objects	138
Alternate is Von Neumann architecture	153	Virtual and pure virtual functions, abstract classes	
Atmel and most microcontrollers - data resides in			

	136	[] referred to as capture, accesses enclosing function's local variables	144
Initializations		What one needs to be aware of	129
Assignment = as initializer	79	Landing point See Exception handling (C and C++)	
Constructor initializes; destructor wraps up	39, 79, 92, 123	Least significant byte (LSB) See Arduino bit functions	
Curly braces {...} as initializer	79	Libraries	
Functions and variables	80	Arduino distribution includes String class	51
Nested array initializations	95	AVRlibc contains most of C Library	51, 99
Parentheses (...) as initializer	79	Previous libraries: Standard C Library, Standard C++ Library, STL (Standard Template Library)	99
Static class variables initialized in global scope	60	Standard libraries not supported by Arduino	150
Use auto to set type and initialize	58	Standard Library extensively documented in The C++ Programming Language	98
Inlining		Standard Library includes previous libraries	99
#define macros resemble inline functions	170	STL (Standard Template Library)	99, 125
Improve performance by inlining functions	142	Linker	
What one needs to be aware of	129	Arduino IDE (tool chain)	20
Insidious bugs See Gotchas (C++ and macros)		Assembles .o files into program	26, 101
int8_t, uint16_t, etc. See Aliases (uint8_t, etc.)		Linker optimizes away unused items	26
Integers, floats, octal, hexadecimal, binary		Linker removes unused items	282
Binary numbers bit-level representation	53	Unable to find a workaround for the Arduino segmentation fault linker problem	27, 199
float contains integer and decimal part, integers do not have a decimal part	52	Unreferenced symbols is undefined variable	26
Hexadecimal numbers start with 0x as in 0xB, octal numbers with 0 as in 013 (decimal 11)	53, 221	Local scope See Global and local scope	
Interoperability		Logical operators	
AtmelStudio/Arduino IDE interoperability	5, 32	&&, , ^ AND, OR operators	66
Interrupts ISR (Interrupt Service Routine)		! ~ negation , also used in bitwise not	65
Attach interrupts and detach interrupts	86	longjmp See Exception handling (C and C++)	
digitalPinToInterrupt -> port interrupt	87	loop See setup and loop	
Internal (software) and external (hardware) interrupts	83, 86	LSB (least significant byte) See Arduino bit functions	
ISR pauses current execution for critical work	86	M	
Port number obtained via INTx	86	Machine code	
Timers	86	bootloader, program stored in flash memory	153
Uno has 2 hardware interrupts, Mega has 6	86	Compiler converts .cpp to machine code .o	25
J		Instructions taken from flash memory put into registers one at a time	153
Java		Macros	
Interpreted OOP language	13	C++ build starts with preprocessing macros	23
K		DELETE and FREE macros release memory and set pointer to null	183
Kernighan and Ritchie		Example macros to handle specific issues	183
Hello World - first C program you wrote	xix	Macros can take parameters	174
The C Programming Language: must read	39	Macros enable customizing the application	12, 172
Keywords		MAX(a,b) takes two parameters	174
Not usable as variable names	25	Multiline macros: see Macros (multiline)	172
Statements (return, break, etc.)	73	Print-based debugging relies on macros	170
KISS principle (keep it simple stupid) See Golden rules and Psychological factors		Text replacements, conditional inclusions, and assemble files together	23, 169, 210
L		Macros (built-in)	
Lambda functions []() {...}			
Creates function on the fly	79		
Ex: auto myLambda = [] (char _abc) {...}	143		

34 | Index table

__FILE__, __FUNCTION__, __LINE__	182	Macros (multiline)	
__func__ has been deprecated	182	/*...*/ and // comments in macros	172, 213
Macros (create)		Backslash for macros across physical lines	172, 212
Backslash for macros across physical lines	171	Do not use // in multiline macros	172
Cookbook presentation on creating macros	171	Forgot backslash in multiline macro	213
Enclose macro parameters and code in parentheses	172	Macros (operators)	
Macros start with a #, lie in single line	171	Concatenation operator ##	177, 178
No spaces after macro's name and opening parens	172	Parameter differentiation operator /**/	177, 179
		Stringizing macro operator #	177
Macros (#define)		Macros (#pragma)	
BUFFER_SIZE example to store a value	174	Sets compiler directive	182
#define can take parameters	174	Macros (#undef)	
#define macro can take parameters	23	Helps create complex macro definitions	176
#define macros resemble inline functions	170	main	
Has 5 parts: # pound sign, type, name, parameters, macro expansion	171	Arduino IDE adds 'main' behind the scenes	44
Simplify source code, improve application's robustness, multiple development scenarios	170, 174	C++ program entry point	40
		setup/loop combination - Why?	44
Macros (#error)		make utility	
#error gets you to look for macro definition problem whereas the macro is fine	181	Arduino IDE (tool chain)	21, 26
Issues compiler error, stops the build	180	Builder is Arduino's name for the make utility	26
#undef to handle complex macros	181	File rebuild dependencies	26
Verify macro coherence via #if logical macros tests and trigger #error	180	makefile - know what you are doing	26
		Prevents duplicate work if file not modified	26
Macros (#ifdef...#elif...#else...#endif)		malloc See new and malloc	
AtmelStudio indents #ifdef, not Arduino	218	Masks See Bit masks	
Comment #endif to match #ifdef	217	Maslow's pyramid	
#ifdef conditionally selects code	175	Physiological and safety needs, esteem, respect, recognition, belonging, self-realization	260
#ifdef where is matching #endif or vice versa	216	Psychological factors	257
Long error messages from unbalanced #ifdef	217	What drives motivation?	259
Macro turns off code due to bad #endif	217	Memory allocations (heap)	
Provide open/close curly braces to find unbalanced/missing #ifdef...#endif	218	Allocation adds two bytes for allocated size	165
Macros (#if defined (...))		Auto allocation variables in stack frame	82, 155
Conditional inclusions	24, 170	Available allocation space too small	160
Create complex logical macro tests using 'defined' and logical operators	175	Careful when allocating memory	247
defined is a macro keyword	175	Check contiguous and fragmented memory	48, 153, 160, 188, 228
Macros (#ifndef...#endif)		Constructor initializes; destructor wraps up	39, 79, 92, 123
#ifndef avoids repeat file inclusions	176, 218	Contiguous heap is top of allocated memory to bottom of stack	158, 160, 166
Macros (#include header file)		Destructors wrap-up chores, release memory	39
Header files (.h) and code files (.cpp, .ino)	24, 176	ExerciseHeap reveals memory consumption	164
#ifndef avoids repeat file inclusions	176	Fragmented memory lies between system memory and contiguous memory	159
Macros (#ifndef...#endif)	176	Function stack frame bottom of stack	155, 166
Pastes the content of a file	176	Global, static, system data bottom physical RAM	159
Macros (logical operators)		Global, static, system data bottom RAM	155
Are macros coherent with one another?	180	Heap: from system zone to stack	158, 164
Logical operator names in clear English	182	Heap grows upward, releases haphazard	154, 160
Macro logical operators AND/OR (&&/)	177, 180	How RAM use evolves	159

Inadvertent bottom of physical RAM write	160	External storage space (RAM, SD card, etc.)	153
malloc is a function, new is a C++ operator	39	Failed allocations, memory fragmented	160
Memory needs using F() macro	189	Getting the address of the bottom of stack	161
new/malloc allocate memory from the heap	39	Memory available for the stack and heap	161
PROGMEM for read-only variables	98, 153, 154	RAM extension chip	153
RAM partitioning	155	SD memory card	153
Stack grabs/releases memory top-down	154, 159	System and user RAM	160
Stack thrashes top of allocated memory	159	Three memory pools: RAM, EEPROM, flash	153
Store read-only variables in flash memory	157	What goes where in RAM - important to know	160
Total heap is fragmented + contiguous heap	160	Memory (total) See Arduino memory pools	
Upon powerup static, global, system data loaded		Methods See Functions (aka 'methods' in OOP)	
bottom physical RAM	155	MicrochipStudio See AtmelStudio	
Use contiguous heap or largest hole in fragmented heap	161	Microcontrollers	
Use two extra bytes in allocated space to verify destination size]Global, static, system data bottom	246	Arduino IDE targets mainly 8-bit Atmel chips	11
Memory corruption		Atmel chips available standalone, Raspberry Pi, ESP 32 on boards	11
Adopt preventive measures	247	Atmel chips based on Harvard architectures	153
Find array size with special end of array value	245	Clocks, ports, timers, interrupts, register size, speed, memory, multitasking, concurrency	10
Invalid pointers	246	Communications protocols	11
Many possible memory corruption causes	160	ESP32, RaspberryPi, IoT	12
Many ways to get into trouble	245	Features which define microcontrollers	10, 15
Memory corruption: where/how to proceed	247	Operating system required?	11
No apparent cause and effect	245	PCs, Macs are Von Neumann architectures	153
Pointer validity not checked	221	Programming languages	11
Print gibberish, wrong values, etc.	157, 245	Reduce development cost with Arduino	19
Runaway index outside allocated space	160	Which chip/language combination?	15
Runaway index thrashes return address	245	Microsoft VBA See VBA	
Stack thrashes top of allocated memory	159	Microsoft Visual Micro See Visual Micro	
String writes, missing end null	246	Microsoft Visual Studio	
Unions misuse can thrash variables	246	C++ Arduino dev with MS Visual Studio	33
Memory (heap contiguous)		C++ Arduino dev with Visual Micro	32
Between fragmented memory and the stack	155, 166	Microsoft Visual Studio See AtmelStudio and Visual Micro	
How to determine contiguous memory	161	millis(), micros(), delay() See Timers	
Memory (heap fragmented)		Min/max values of built-in types	
Allocation space too small - fragmented heap	160	Table of numeric types	54
__brkval top fragmented heap, __flp bottom	164	Mistakes concerning this book See See Books Web site (md-dsl.fr)	
Fragmentation ratio	166	Mixing numeric types See Gotchas (C++)	
Fragmented memory, sum total of holes, measured by traversing list of holes	158, 165	mnemonics	
Memory releases are haphazard hence heap is Swiss cheese like	155, 165	typedef	57
Total available memory fails to disclose highly fragmented memory	160	Modularization See Linker	
Memory (pools)		Most significant byte (MSB) See Arduino bit functions	
Flash memory is where the program (.hex file), PROGMEM variables, F() macro strings reside	27	Motivation	
Three memory pools: RAM, EEPROM, and flash	27	Maslow's pyramid	259
Memory pools See Arduino memory pools		Psychological factors	273
Memory sharing See Unions		What drives motivation?	259
Memory structure		MPLAB	
		May be overkill for Arduino dev	34

MSB (most significant byte) See Arduino bit functions	new/malloc pointer to allocated space 39, 162, 227
Multidimensional arrays See Arrays and indices	Use two extra bytes in allocated space to verify destination size 246
Multiline macros See Macros (multiline)	
Multiple inheritance See Inheritance	
Multitasking and concurrency	new overloading
C++ features not supported by Arduino 151	Error checking via overloaded new 162
C++ short history 9	Overloaded new implements malloc 162
Concurrency: run several tasks in parallel 151	size_t _allocSize to define allocation size 162, 163
Go programming language 13	Use C exception handling (setjmp/longjmp) to handle errors detected via overloaded new 162
Interrupts enable doing basic multitasking 87	Non-volatile memory
Multitasking means run two or more tasks, concurrently or on a time-shared basis 151	EEPROM 154
Multitasking requires OS to swap tasks 151	Flash memory 154
Multitasking: run two or more tasks, concurrently or time-shared 11	NOT ~ and ! See bit-level and logical operators
mutable See const and mutable	Notepad++
	Good programmer multi-language editor 21
N	NULLPTR
Name completion	Variable initialized as null pointer 81
Arduino 2 supports name completion, not 1.9 22	null statement
AtmelStudio, Visual Studio, PlatformIO support name completion 22	';' null is simplest possible statement 49
Name mangling See Functions and variables	Solves no code after label problem 252
namespace	Numbers See integers, floats, octal, hexadecimal, binary
C vs. C++ 38	Numeric constants See Constants - character, string, and numeric
Grouping data together in namespace (class) key to user-defined types top-down programming 38, 144	O
Keyword 'using' C++ namespace equivalent 145	Objective C
namespace: use variables and functions with same name from different libraries 144	Early object-oriented programming language 8
Scope operator :: resolves name clashes 38, 144	Object-oriented programming
What one needs to be aware of 129	C++ classes and structures 91
Nesting	C++ mechanics 43
Array initialization 124	C++ short history 8
Curly braces define local scope 104	How one thinks (top-down) 8, 89
Recursive functions call themselves 165	Simula, Lisp, Objective C, Smalltalk 89
Never assume anything	Types (user-defined) 9, 17, 56
Error handling 145	What defines C++ 17
Good programming practices 47	Which chip/language combination? 15
new and malloc	Octal numbers See integers, floats, octal, hexadecimal, binary
Allocate memory with new or malloc? 161	OOP See Object-oriented programming
Careful: new int() and new int[] different 231	Operator overloading
Customize new and delete (overload them) 162	C++ mechanics 101, 129
DELETE (new) FREE (malloc) release memory & set pointer to zero 183, 246	Complex numbers use overloaded operators 130
Gotchas (C++) 227	Operator overloading customizes operator 130
Memory allocations (heap) 65	Overloaded index operator[] 101, 131, 146, 276
new/delete operators, free/malloc functions 161	SafeArray template class 276
::new - global new 163	Operator precedence See Precedence and associativity
new int() vs new int[] completely different 219	Operators
	Build expressions and statements 64
	C++ building-blocks 49

new/delete operators, free/malloc functions	161	References resemble dereferenced pointers	118
Precedence and associativity	65, 67	Saves RAM when copying objects	117
sizeof is an operator	65	Work on external function variables	115, 119
Table - C and C++ operators	65		
try is an operator	73		
OR and See bit-level and logical operators			
Order of execution See Precedence and associativity			
Ordinary arithmetic vs. pointer arithmetic			
Access array items via subscript or pointer	124		
Dereferencing a pointer	219		
Index and pointer-based array traversal	280		
Index- and pointer-based array traversal	124, 280		
Invalid char(*)[4] to uint16_t	201		
Ordinary arithmetic using sizeof(char*)	281		
Plain and pointer arithmetic differ	124		
Pointer arithmetic using sizeof(char*)	281		
Size of items in an array	280		
Other IDEs			
AtmelStudio, Visual Studio, Visual Micro, PlatformIO, Code::Blocks	33, 34		
Other IDEs See Development tools			
Out-of-bounds index See Runaway index			
out_of_range			
Exception class events	149		
Overflow_error			
Exception class events	149		
Overflow/underflow during expression evaluation			
Careful with precedence/associativity	224		
Compiler upgrades expression to 16 bits	225		
Improper data sizes in expression evaluation	224		
P			
Parameter differentiation operator /**/			
Enables parameters be joined	179		
Parameter passing			
C++ mechanics	42, 102		
Careful when passing array as parameter	57		
Careful with type checking leniency	121		
Three ways: pass by value, address, reference	117		
Pass by address			
Address of variable via operator &	114		
Dereferencing is a computerese trick	114, 117		
Pass by address to modify external variable, pass by reference is alternate	114, 115		
Pointer parameters	114		
Pass by reference			
Alphabetical sort on a contact list saves RAM	117		
Bitfields may not be referenced	119		
Dual nature of a reference	118		
Pass indirectly with a reference cast or directly with a reference variable	116		
Reference initialization	118		
		Pass by value	
		Function creates copy of parameter	112
		Parameter can be a constant, variable, function, function pointer, Lambda function	113
		Simplest parameter passing mechanism	112
		PCB (Printed Circuit Board) See Arduino tool chain	
		Permanent storage	
		EEPROM for read-write variables	154
		Permissive flag (compiler leniency)	
		Compiler options	25
		Invalid char(*)[4] to uint16_t	202
		This flag is required by the Arduino IDE	25
		Transforms errors into warnings	25, 202
		Type checking leniency	234
		PGMP	
		Convenience PROGMEM macro	190
		pgm_read_byte and pgm_read_word	
		PROGMEM framework	187
		PHP	
		Mainly used for Web server applications	13
		Pin or port	
		Port for boards, pin for microcontrollers	86
		PlatformIO	
		Debugging (hardware-based) for ESP32	34
		Extensive learning curve	33
		Plug-in tool for Microsoft's VScode	33
		Supports name completion	22
		platform.txt	
		Arduino configuration file	25
		-flto flag (link time optimization)	25
		-fno-exceptions flag	147
		-fpermissive mandatory for Arduino dev.	25
		Pointer	
		Array as parameter converted to pointer	57
		const pointers offer interesting possibilities	61
		new/malloc should return non-zero pointer	162
		Pointer-based array traversal	280
		Pointer differs from index yet interchangeable	57
		Pointer such as int myVal* (pointer to an int)	57
		Type qualifiers	79
		Using a pointer directly without assigning memory space creates phantom object (bug)	219
		Pointer arithmetic vs. ordinary arithmetic See Ordinary arithmetic vs. pointer arithmetic	
		Polymorphism	
		Derived class redefines base class virtual functions	120
		Means take on many forms	120
		Name mangling differentiates functions	120

38 | Index table

Port or pin			
Port for boards, pin for microcontrollers	86	Store read-only variables in flash memory	157
Ports See Arduino specific functions		Store struct and class data in flash memory	194
Postfix See Prefix/postfix operators (++/--)		Storing an array of strings in PROGMEM requires special handling	192
Powerup		strcpy_P loads string from flash memory into RAM buffer	187, 193
Global, static, system data bottom RAM	155	Things not to do with PROGMEM	190
Memory allocations (heap)	155, 159	Web sites which describes PROGMEM	286
System loads instruction from flash memory address 0	155	Program components See C++ components	
Pow() problem		Program execution	
Add 0.01 to the result of pow()	239	Instructions taken from flash memory put into registers one at a time	153
Pragmatic C++		Programming languages	
Introduction	1	Bottom-up programming	10
Practical approach to programming C++	1	C, C++, C# (C-sharp)	12
What this book is all about	xx	C programming	10
Precedence and associativity		Delphi - today's version of Pascal	14
+ higher precedence than shift (<< >>) & ==	242	Fortran	14
Associativity: right to left, left to right	64, 68, 101	Go programming language	13
C and C++ operators	65, 67	HTML (Hypertext Markup Language)	13
Can you spot the problem in 5 == 5 + 2 == 3	215	Java, JavaScript, PHP	13
Dereferencing a pointer has low precedence	219	Programming languages for Arduino	12
*myPointer++ Is it (*myPointer)++ or *(myPointer++)?	225	Python and WiPy	13
Precedence first, associativity next	64, 68, 101	Smalltalk and Objective C early OOP language	8
Use parentheses to define evaluation order	225	VBA (Microsoft's Visual Basic for Applications)	13
Values during expression evaluation	68	Program See Executable file	
What gets done first, add or multiply?	65, 67	Prototype board See Arduino	
Prefix/postfix operators (++/--)		Pseudo-exception handling See Exception handling	
Prefix/postfix increment/decrement	65, 106	PSTR() macro (PROGMEM)	
Prefix/postfix operators are misunderstood	106	F() macro saves RAM on Serial.prints	188
Preprocessor See Macros		PSTR() macro stores variables in flash memory	189
Print-based debugging See Debugging (print-based)		Psychological factors	
private See public		10 commandments of EGOless programming	260
Procedural programming		Cognitive dissonance and EGO	258, 271
Bottom-up programming - no top-down	8	Flaw in my thought process	266
C++ short history	8	Good and bad habits	258
Characterized by functions which process data	89	Hobbyist's mindset	257
Chasm between code and thinking	89	How to be an efficient programmer	258
gotos extensively used lead to spaghetti code	12	I code therefore I am	264
Many gotos generates spaghetti code	76	Incremental learning	268
Structural programming - gotos banned	17, 76	Incremental vs. planned programming	259
PROGMEM (read-only data in flash memory)		KISS principle: or why complicate things	266
C++ building-blocks	50, 98	Know yourself - listen to your body	258
F() and PSTR() macros save RAM by storing data in flash memory	98, 153, 185, 189	Leave the sandbox to solve the problem	266
PGMP is a convenience macro	190	Maslow's pyramid	257, 258
pgm_read_byte and pgm_read_word	187	Memory, responsibility, patience... influence programming performance	271
PROGMEM qualifier store in flash memory	187, 193	Motivation	273
Small program tests PROGMEM storage	275	Planning work offline important	259, 264
Store debugging messages in flash memory	191	Programming requires a healthy mind	257
Store __FILE__ in flash memory	83, 191	Psychologically induced errors	259
Store float values in flash memory	194	Psychology of computer programming	258
		See ChatGPT - one more tool to help you write	

better code	268	Overload operator [] protect from bad index	160
Thinking is the hardest thing to do	259, 265	Runaway index thrashes return address	159, 245
To err is human	270		
Why does one make mistakes?	208, 263		
public		S	
Class data visible from the outside	135	SafeArray class	
private class data not visible from the outside	135	Array [] type qualifier and index [] operator	131
private for data NOT visible from the outside	135	Exception handling (C and C++)	146, 279
Pure virtual functions See Virtual & pure virtual functions, abstract classes		Handling runaway index	279
PWM pins See Arduino specific functions		Index [] operator overloading	146, 277
PWM (Pulse-width modulation)		SafeArray template for custom typed arrays	277
PWM is used to create analog-like voltage	85	Template for custom typed arrays	276
PWM (Pulse Width Modulation) See Arduino specific functions		Scope resolution operator ::	
Python and WiPy		Accessing class items outside the class	69
Arduino IDE (tool chain)	16	Items in a namespace	69
Interpreted language	13	Items in an enum list	69
Which chip/language combination?	14	Scope (visibility)	
R		C++ is a highly scoped language	18
RAM (Random Access Memory) See Memory structure		C++ mechanics	101
Random numbers		class and struct	106
Arduino specific functions	83	Curly braces encapsulate switch cases	106
Hardware driven random numbers	88	Function masks variable in outer scope	105, 219
range_error See Exception class events		Nameless encapsulation improves readability, saves RAM	106
RaspberryPi		Scope levels (curly braces nesting)	105, 106
Which chip/language combination?	14	Variables have global or local scope	18, 104
Real world entities are the way one thinks See C++ short history		Scratch for Arduino See Visual development	
Recursion See Stack and stack frames		SD memory card	
Recursive functions See Nesting		Memory structure	153
Reference & type qualifier		Segmentation fault See Linker	
Reference an object, no need to dereference	56	selective code inclusion	
Reference & type qualifier		Use #ifdef macros for print-based debugging and alternate dev scenarios	210
Address of or reference to (contextual)	116	Semantics (vocabulary)	
Reference as int myRef&	57	Grammar (rules)	25
register See const and mutable		Sequence expressions	
Regular expressions (regex)		comma operator	65
Immensely useful tool not supported by Arduino editor, supported by AtmelStudio	151	Serial communications	
See Defensive C++ Arduino Programming book	151	DEC, HEX, OCT, BIN Serial.print modifiers	88
Reset See Powerup		Serial.begin, Serial.end, if(Serial), etc.	88
Return address See Stack frames		Serial class used for serial communications	87
Return value See Functions and variables		Serial.print modifiers	88
runaway index		Serial.print, Serial.read, etc.	88
SafeArray class overloads index operator []	245, 279	Serial terminal	
Runaway index		Arduino IDE (tool chain)	21, 29
Gotchas (C++)	245	AtmelStudio	29
		PlatformIO	29, 34
		Visualize program behavior via Serial.prints	29
		setjmp/longjmp See Exception handling (C and C++)	
		setup and loop	
		Arduino specific functions	44
		Do a while(true) in setup instead of using loop	44

40 | Index table

loop for event-based programming	45	Control flow statements (if, while, etc.)	72
setup runs once, loop runs indefinitely	40, 44	End with a semicolon	49
Signal functions		Fundamental program units accomplish work	72
Arduino functions to manage signals	83, 85	Program is a collection of statements inside curly braces code-blocks	49
pulseIn(), pulseInLong(), shiftOut()	85	static	
Square wave, frequency, duration	85	class variable to be instantiated once only	58
tone() and noTone()	85	Functions and variables	80
signed or unsigned See Types (built-in)		Global, static, system data bottom RAM	155
sizeof operator		Old C style and new C++ usage	58
Defines memory needs of an object	66, 70	Static class variables initialized in global scope	60
Don't forget, sizeof is an operator	70	static_cast See Enumerations	
Getting size of object is fraught with gotchas	230	static data See Global scope	
sizeof array - no. of items or memory needs?	70	STL (Standard Template Library) See Libraries	
size_t		Storage specifier See Declarations	
Address size of the target system	64	strcpy_P See PROGMEM	
Atmel 8-bit chips have 16 bit addresses	64	String constants See Constants - character, string, and numeric	
ESP32 has 32 bit addresses	64	Stringizing macro operator #	
Sketch (.ino file) See Arduino C++ editor		Macro prints variable's name instead of value	177
Smalltalk		Strings	
Early object-oriented programming language	8	Arduino distribution includes String class	51
Source code See Book's Web site		Break up long strings into substrings	234
Spaghetti code		C++ mechanics	42, 102
Excessive goto use generates spaghetti code	12	char* array init fail missing comma	235
GOTO wreaks havoc in procedural programs	73	Constants - character, string, and numeric	52
Many gotos generates spaghetti code	76	Length functions do not include end null	122
Procedural programming rely on gotos	12	Linker discards unused strings	192
Special character escapes See Escape sequences		Missing end null cause of thrashed memory	240
SRAM (Random Access Memory - RAM) See Memory structure		Often-used pre-defined strings	191
Stack and stack frames		Pros and cons of char strings vs. String class	122
Contiguous memory below bottom of stack	166	Several ways to manage read-only strings	191
Functions should contain explicit returns	235, 236	Store strings in flash memory with F() macro	185
Memory allocations (heap)	166	strcpy, strcat, strlen, strcmp functions	42
Monitor function call memory requirements	161	Strings in bottom of RAM at startup	155
Out of contiguous heap space	159	strlen(myCharStr) & myStr.length() functions	122
Runaway index thrashes return address	160	Use PROGMEM's F() and PGMP() macros	191
Simple recursion function stack frame size	167	Strings (zero-based indexing forgotten)	
Size of stack frame	167	Array dimensioning macros will save you time	241
Stack frames added at bottom of stack	161	C++ arrays are zero-based	82
Stack frames contain function parameters, variables, return value & address	155, 167	char strings end with a null	240
Stack frames function call overhead	167	N-size array, first at index 0, last at index N-1	239
Stack frames last-in/first-out (pushed and popped from the bottom of stack)	158	Usually first means one, not in C++	239
Stack grabs/releases memory top-down	154	Strong type checking	
Stack located in top of RAM	158	C++ short history	9
Stack thrashes top of allocated memory	159	Type checking leniency	232
Standard Library, Standard C Library, Standard C++ Library, Standard Template Library (STL) See Libraries		What defines C++	17
Startup See Powerup		Structural programming See Procedural programming	
Statements		Structures See class and struct	
		Subscripts See Arrays and indices	
		switch (value) {...}	

Control flow statements	74	class and struct	139
Default case not mandatory, should exist and contain code	76	Linked list	139
Encapsulate cases in curly braces	76, 106	throw	
Execution choice (case) based upon criterium	76	throw operator in C++ exception handling	65
Synonyms See Aliases		throw See Exception handling (C and C++)	
Syntax differences between C and C++		Timers	
C allows global duplicate declarations	45	Arduino specific functions	83, 85
C and C++ handle goto differently	46	delay() and delayMicroseconds()	85
C does not support name mangling	45	Interrupts	86
char constant is 16 bits in C, 8 bits in C++	46	millis() (50 days limit) & micros() measure time from startup	85
Declare a struct within a struct (nesting)	45	Specify delay in while statements	76
Declare a struct within a struct (struct nesting)	45	Tool chain See Arduino IDE (tool chain)	
System data See Global data		Top-down programming	
system_error See Exception class events		Bottom-up characterizes 3rd gen. languages	10
System variables		Bottom-up programming handles details, top-down maps how you think	43, 89, 91
__brkval top fragmented heap, __flp bottom	164	C++ mechanics	12, 40
__FILE__, __FUNCTION__, __LINE__	83	Object-oriented programming is top-down	8
__flp/__brkval determine fragmented memory	83	Types (user-defined) is top-down	56
__func__ (deprecated)	83	What defines C++	17
T		Total memory See Arduino memory pools	
Tables		Transmission constraints	
Data packing into bitfields	132	Work-around to send data which excludes 255 values	284
Date and time packed data	133	Traps and pitfalls See Gotchas (traps and pitfalls)	
Declaration examples	79	try See Exception handling (C and C++)	
Flash memory with & without F() macro	189	Type checking	
Flash, RAM, and EEPROM sizes	154	auto type qualifier	219
GitHub on for Arduino, RaspberryPi, and ESP32	16	char* & ints compatible, char* & float NOT	232
How RAM use evolves	159	Extreme type checking not practical	121, 232
Memory needs using F() macro	189	-fpermissive flag (compiler leniency)	234
Min/max to exclude 255	284	Gotchas (C++)	219
Min/max values of built-in types	54	Missing parameter creates havoc	233
Numeric types (C and C++)	54	Type checking leniency in parameter passing	102, 121, 233
Operators (C and C++)	65	Type checking leniency See Types (equivalent numeric types)	
Type qualifiers (C and C++)	56	typedef	
Templates		Simplifies programming by creating aliases	57
Advanced mechanisms	125	typedef	
C++ mechanics	102	Declarations	78
GetMinOfTwo - function template example	125	Function pointers	140
Operator overloading	276	Typedef FPtrVoid fpHelloWorld	141
SafeArray class template	277	Type qualifiers	
STL (Standard Template Library)	125	Array type qualifier []	79
T based template class	128	C++ building-blocks	49
Type independent classes and functions	102, 125	C++ program components	56
WolfPack template class example	127	Derived type from type qualifier on basic type	56
Terminating null See Strings		Function () type qualifier	79
The C++ Programming Language book		Lambda []() creates function on the fly	79
14 chapters describe the The Standard Library	150	Pointer *	79
Bjarne Stroustrup	150		
Preface	xix		
this			
Address of current object, refers to self	139, 140		

42 | Index table

Table of type qualifiers	56	VBA	
volatile and register	63	Evolved from early BASIC (DOS)	10
Types (basic and derived types)		Word, Excel, PowerPoint, etc. programming	13
Aliases (uint8_t, etc.)	54	Verbose messages See Error messages and warnings	
auto built-in type automatically sets type	55, 79	Virtual and pure virtual functions, abstract classes	
Basic type + type qualifier ==> derived type	53, 56	Polymorphism	120
Built-in type sizes, from 1 byte to 8 bytes	54	Virtual & pure virtual functions, abstract classes	
C++ building-blocks	49	Derived class redefines base class functions	136
char types are signed - don't know why	55	Pure virtual functions: no code	136
Declarations	78	Virtual functions key to inheritance flexibility	136
int, long, char, etc. are built-in types	54	Virtual & pure virtual functions, abstract classes	
Numeric types (compatible)	121	Pure virtual functions: no code	138
Serial.print cannot print individual extended ASCII		Transparently cycle derived classes	137
characters (ex. ñ) but can in string	55	Visibility See Scope	
size_t returns microcontrollers address size	55	Visual Basic for Applications See VBA	
unsigned modifiers, signed by default	54	Visual development	
User-defined types enrich basic types	56	Blynk for Arduino - business oriented tool for IoT	
Types (equivalent numeric types)		devices	35
C++ mechanics	102	Drag and drop code-blocks	35
Mixing integer and pointer parameters gotcha	121	Scratch for Arduino - tool designed for children	
Numeric types equivalence concept	121	who want to program	35
Strong typing not carried to extremes	121	Visualino - promising visual development, stopped	
Type checking leniency vs. strong typing	121	evolving in 2017	35
Types (user-defined)		Visualino See Visual development	
C++ short history	8	Visual Micro	
class and struct	56	Provides the upload process to Visual Studio	27
Create programs close to how you think	43	Supports name completion	22
Object-oriented programming (top-down)	17, 56	Supports serial debug	29
Simplifies programming (map how you think)	56	void functions	
What defines C++	56	Specify a return type otherwise returns int	79
U		void function does not return anything	79
uint8_t, int16_t, etc. See Aliases (uint8_t, etc.)		volatile See const and mutable	
Underflow_error		Voltage level	
Exception class events	149	Digital and analog I/O functions	84
Underflow See Overflow/underflow in expression evaluation		Von Neumann architectures	
Undo (ctrl-Z) bug See Arduino IDE bugs		Alternate is Harvard architecture	153
Unions		PCs, Macs program and data reside in RAM	153
Beware, a union looks like a structure	96	VSCode See PlatformIO	
C++ building-blocks	50	W	
Decompose float into 4 bytes to send across a		Warnings on	
serial port	96	Arduino IDE	224
Save RAM via memory sharing	96	Errors become warnings thus allowing build	202
Union member thrashes other member	96, 246	platform.txt	202
What defines C++	18	Web site See Book's Web site	
Unreferenced symbol See Linker		What defines C++	
uploader See avrduide		C++ is a highly scoped language	18
User-defined types See Types (user-defined)		C enhanced with user-defined types	16
V		class and struct	18
Variables See Functions and variables		Create compact, fast applications	16
		Direct memory access	18
		Embedded C	16

Inheritance	17	X
Object-oriented programming (top-down)	17	XOR ^ See Bit-level operators
Rich set of operators	18	
Strong type checking	17	Z
User-defined types	56	Zero-based indexing forgotten See Strings (zero-based indexing forgotten)
What is a C++ program		
Complete minimalist Arduino program: main (hidden) + setup + loop	40	
Modularization (collection of .cpp modules)	40	
Starts with main, calls other functions	40	
What one needs to master	37	
What one needs to be aware of		
C++ features not supported by Arduino	130, 150	
Complex numbers	130	
Data packing (bit-level)	129	
Error handling and exception handling	129	
Functions and variables	129	
Inlining	129	
Lambda functions <code>[](){...}</code>	129	
namespace	129	
The C++ Programming Language book	150	
What one needs to master		
Arduino specific functions	83	
C++ building-blocks	2	
C++ C++ mechanics	2	
C++ enhancements to C	37	
Libraries	51	
What is a C++ program	37	
Which chip/language combination?		
Arduino in short list	14	
Choosing programming language & target micro-controller is compromise	14	
Costs and development time	15	
Devices, sensors, hardware support	15	
Employment and futureproof knowhow	15	
ESP 32, RaspberryPi, Espruino, WiPy	14	
Python and WiPy	14	
while (condition) {...}		
Control flow statements	74	
while to for transformation	75	
while (true) in setup() replaces loop()	44, 75	
Why does one make mistakes?		
Being careful is not good enough	207	
Inadequate offline preparation	208	
Poor physical and/or mental condition	208	
Psychological factors	208	
Tweak and test the algorithm	208	
Why I wrote this book		
Programming my beehive weighing system was a hassle	xix	
WiPy See Python		
Word boundary See Bitfields		

PRAGMATIC C++ ARDUINO PROGRAMMING

Understand C++, discover its many gotchas, develop a professional mindset.

If you find that you are dedicating much too much time developing C/C++ applications and experience it as a hard, grueling task, this book is for you.

There are *good reasons* to use Arduino and C++ instead of another microcontroller and Python or other programming language.

Concentrate on what one needs to master – i.e., what we, Arduino programmers, need to know concerning C/C++, and what we can avoid spending time on yet should be aware of.

Don't just know C/C++, understand it! Many of C/C++'s unique features (parameter passing, pointer arithmetic, referencing, ...) are explained; extensive code examples clarify concepts.

C++ being deceptively simple, discover the many gotchas it can throw at you.

Memory management – know how memory gets used and monitor it: avoid *stack overflows* and *out of memory conditions*. Understand what may corrupt memory and anticipate its use.

Discover *C/C++'s preprocessor*, a unique extremely practical feature practically no other language has. Use it way beyond simple `#defines`. A dedicated chapter covers the *preprocessor*, its syntax, how to use its macro mechanism, and how to avoid its many gotchas.

Save RAM by storing *read-only variables* in flash memory via the PROGMEM framework. Store read-only float values and the contents of read only struct and class data. Do's and don'ts are presented with concrete examples.

Use *operator overloading* to simplify programming and improve on the application's robustness. The overloaded new operator enables memory management; the overloaded index operator [], used within the context of a `SafeArray` class, prevents under or overshooting array boundaries.

Handle errors with `setjmp/longjmp`, an alternative to the unsupported C++ exception handling.

Psychological factors which influence productivity are covered. One's mental condition, good vs. bad habits, understanding Maslow's pyramid hierarchy of needs, are keys to productivity.

The companion book, *Defensive C++ Arduino programming*, introduces *AtmelStudio* and *Visual Studio 2022* plus *Visual Micro* to develop Arduino applications with. It also introduces *Awk*, *Perl*, *regular expressions* and proposes *frameworks* to quick start your application.

Download free open source licensed frameworks source code and *Awk*, *Perl*, an *regular expression* tidbits - download link in <https://md-dsl.fr>.

Being pragmatic means do whatever it takes to obtain results: adhere to good programming practices, hone your C/C++ skills, understand how psychological factors influence the quality of your work.



9 782958 562809

Kindle \$3.85

Paperback \$27.85

Hardcover \$24.85

Marmosset drawing by the author