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



First published via Amazon August 2023.

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


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

*This page intentionally left blank*

# List of tables and figures

*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 (4$^{th}$ 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 programing 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 `int`s, `char`s, `float`s, 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 IofT 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:



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, Forth, 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

## D

## E

# 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 o*perator 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*