Use Visual Studio with Visual Micro, hardware debug with AtmelStudio (MicrochipStudio).

DEFENSIVE C++ ARDUINO PROGRAMMING

Self-published via Amazon by the author - Michèle Delsol

Defensive C++ Arduino Programming by Michèle Delsol

First published via Amazon August 2023.

Delsol, Michèle
Defensive 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 warthog illustration on the front cover is an original pencil and China ink drawing by the author.

Paperback edition : ISBN 978-2-9585628-2-3
First edition published via Amazon August 2023 in paperback, hard cover, and electronic (Kindle) formats.

Table of contents

List of tables and figures

Acknowledgements

This book, *Defensive C++ Arduino Programming* and its companion, *Pragmatic 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 accepted 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.

Preface

How did I come to write *Pragmatic C++ Arduino Programming* and *Defensive C++ Arduino Programming*? The journey started due to my being an amateur beekeeper. In order to check the health of my hives, I weigh them regularly, weekly, or daily, depending on the season. I use an ordinary luggage scale to lift the rear of the hive. I naturally asked myself "How about hacking an electronic *Arduino-based beehive weighing system* to replace my trips to the beehives. Monitoring the weight provides an indication on the beehive's health and how much honey is being produced. It also helps determine whether the beehive has swarmed. Swarming means that the old queen and half the bees leave the beehive; a 6 lbs. loss in 10 minutes.

The back of the envelope design of the *Arduino beehive weighing system* seemed simple. Building such a system was the type of challenge I would enjoy. Little did I know that the electronics would prove easy, while the software to control the electronics would be a major endeavor which would take two+ years to complete! As I progressed on my project, I was forced into transitioning from a hobbyist mindset to a professional programmer mindset. These two books are the concepts, procedures, and tools which enabled me to complete the *beehive weighing system software* – its size is now 35 files/15000 lines of code.

I quickly defined the general characteristics of the project: ordinary bathroom scale strain gauges amplified by opamps, star shaped network radio communications, clocks to synchronize the modules, a GSM board to send data to my smartphone. The deciding factors on choosing components were cost, availability, suitability, and ease of development. The inexpensive Arduino board, its free easy to use development environment, and C++ being the best language for producing compact, fast, executables were the deciding factors. What's more, I already had experience programming C++.

I then wrote code to try out the subsystems one at a time. Throughout this early dabbling, I was impressed by the ease provided by the Arduino IDE. After experimenting with the examples, I started my *beehive weighing system* with one *.ino* file, then, slowly but surely, the application grew. I added *.h/.cpp* files, created classes, and defined macros.

I started the project by validating my choice of electronic components and experimenting with each subsystem. And this is where the fun began. Getting subsystems to work one at a time was easy. Getting them to work together got to be particularly time consuming. I confess that I had unknowingly embarked on a major project with a hobbyist mindset. I soon spent much more time debugging than coding. It got to be a painful experience.

As I progressed and the application was getting more complex, I was at the mercy of poor planning, and I no longer controlled development. I could not continue with my undisciplined hobbyist approach hacking out an *Arduino-based* gadget by coding directly, straight from mind to code. I had to modify my ways because the application was getting unwieldy. It had reached a size and complexity which required that I adopt formal design and implementation techniques. If I were to bring my project to completion, I had to step back, rethink my approach, and get organized. Furthermore, I had to update my C++ knowhow since I had become a little rusty and was making way too many mistakes.

As luck would have it, I had already developed applications at a professional level and had

been project manager of a large industrial plant project, but that was years ago. I thought back on that work and reviewed literature on the subject. My past experience incited me, slowly but surely, to adopt good programming practices and to structure development. This proved to be difficult and challenging. It demanded hard work, but the fun returned.

The foremost change in my approach was to formalize the parts and pieces of the project, define them as individual modules, and lay them down on paper. Understanding the system's structure from both the electronic and software standpoint led to partitioning the project into modules. There were subsystems at the electronic level (GSM, clock, radio, EEPROM) and there were subsystems at the software level (data storage, data validation, data conversion, data transmission, etc.). Thanks to C++'s object-oriented features, I started the application's design *top-down*. The details were to be filled-in *bottom-up* later on.

Informal solutions to manage certain tasks morphed into formal frameworks. Getting these done required drawing flow charts, creating algorithms, defining functionalities, components, and procedures. Working offline on paper proved to be a critical first step. This planning turned out to be vital to the application's robustness, as well as to my own productivity. After a while, the application began to take shape as pieces fell into place.

Development became a two-pronged process: *design and implement the big picture* (i.e., *top-down* via classes and wrapper functions) and *design and implement low level helper functions and functional frameworks - bottom up*. The process was iterative: go to the drawing board; work on a module; change things; go to the computer; implement new code; test and examine results; then go back to the drawing board. I did this until I got it right. Documenting each step helped me learn and clarify concepts. Many hours later, I had laid the groundwork and developed standards, procedures, and frameworks. In the process of doing this, I documented code as much as possible for future reference.

I took notes on the causes of the various bugs I was getting (array overruns, forgetting the end null in `char` strings, etc.). Then I created frameworks (file organization, register storage of small values, extract documentation out of source files). I jotted down items of interest as I progressed to hone my understanding of C/C++ features: adopt practical how-to procedures; create frameworks to accomplish specific tasks; learn how to use AtmelStudio; use regular expressions to facilitate searches; use a *Perl* program to produce product documentation, one of many tasks *Perl* could help me with. It then dawned on me that there were amateur Arduino developers, not professional C/C++ programmers, who could benefit from this work. I could write a *pragmatic presentation* of the C/C++ know-how I had learned. This morphed into *Pragmatic C++ Arduino Programming*. The *defensive procedures* and *frameworks* I had developed became *Defensive C++ Arduino Programming*.

Writing these two books turned out to be as much of a challenge as designing and creating my *Arduino-based beehive weighing system*. Little did I suspect, when I started putting together the electronic components of my system, that it would take me two years to get the software done, and that I would write two books to convey these newly acquired skills.

Knowhow, good practices, good tools, good frameworks are the keys to success. I should add hard work to this list; success does not come free. Adopting these concepts and tools will make you more efficient: avoid C/C++ gotchas, manage memory, make the program efficient (small and fast), facilitate debugging, understand what you wrote months past because you commented your code correctly, build in safeguards to protect the application as it unravels, modify your code painlessly, etc.

Introduction

I wrote this book with the intent that it should help you, a DIY C/C++ Arduino programmer, evolve from an amateur mindset condition into a professional one, so that you could undertake large complex projects. Time spent adopting good practices, learning how to use professional tools, and implementing frameworks will be amply compensated by less debugging time, much improved adaptability to change, faster more compact code, more robust applications, and a more enjoyable programming experience.

- *Productivity* - Get the project out faster with less pain. Programming should be fun.
- *Maintainability* - Modifications should be easy. Logic should be clear and simple. Complexity should be reduced via encapsulation and frameworks.
- *Robustness* - The application should be free of crashes. This means not using bad pointers, avoiding runaway memory condition leading to stack overflow or heap exhaustion, and staying clear of other such C/C++ mishaps. You will need to hone your C++ skills. The companion book, which dwells deeply into the why's and how's of C++, *Pragmatic C++ Arduino Programming,* could help you with this.
- *Compactness and speed* - Monitoring code size is important because there is a premium on RAM. You may also need to monitor application speed if this were to become critical.

Writing small code which fits in a single sketch file differs from writing large code which needs to be spread across files. The latter requires planning, workflow, tools, and a corresponding mindset. If the application is small (one *.ino* sketch file) you could hack it out directly - mind to keyboard: *incremental programming*. However, if the application is large, you should adopt good programming practices, plan and implement procedures and frameworks, and use professional tools. I refer to this as *defensive programming*. If you do not evolve from a *hobbyist mindset* to a *professional mindset*, you will spend ages trying to get your application to work, the danger being that you might never get it finished.

This book is a compilation of lessons learned and frameworks I developed through two years of experience developing my *Arduino-based beehive weighing system application* (35 files/15000 lines of code). It covers the Arduino IDE (1 and 2), AtmelStudio, Visual Studio, Visual Micro, PlatformIO, VS Code, Code::Blocks, and MPLAB (the last two superficially). It also covers debugging, program organization, program documentation, frameworks, etc. Whichever IDE you use, you can benefit from being disciplined as you work on your project and use frameworks to manage various functional domains.

The chapter *Common sense and new tools* (page 1) should be read first. It presents a concise summary of the content of this book. The chapters *Good programming practices* (page 1), *Frameworks* (page 1) and *Debugging* (page 1) should be read next, they build on each other. The remaining chapters and the *Appendix* (page 1) are self-explanatory. They can be read in random order. They will most certainly influence your development choices.

The main chapters of this book are:

- *Chapter 1* - *Common sense and new tools* (page 1) summarizes the main subjects covered in this book.
- *Chapter 2* - *Why choose Arduino* (page 1) - You chose Arduino as your core hardware, as

opposed to ESP32, RaspberryPi, or another microcontroller. For my *beehive weighing system*, Arduino with C++ was the best compromise.

- *Chapter 3* - The *The build toolchain* (page 2) - You have been using the Arduino IDE since you first discovered Arduino. It is a great tool. As this book is going to print, Arduino just released version 2 of its IDE, an improvement over the one we have been using (latest 1.8.19). Version 2 did manifest a few problems; I feel that it is not solid enough to fully replace the old version (referred to as legacy Arduino). I dedicated a section to version 2 (*Arduino IDE V2 specifics* page 2) and covered version 1 more extensively (*Arduino IDE* page 2) .
- *Chapter 4* - *Hardware setup* (page 2) - Programming your chip is one thing. Getting it to drive the electronics is another thing. Initial experimentation is usually done on breadboards. You ultimately reach a point where the jumble of wires proves unsatisfactory. This is when it is time to get the soldering iron out and migrate to prototype boards. When the application runs properly and the design has been validated, go for your own PCB with its own Atmel microcontroller. You may ultimately develop a final professional grade SMT (surface-mount technology) PCB.
- *Chapter 5* - *Interoperability* (page 2) - The Arduino IDE is a fantastically easy to use tool. But, you will find that you will be much more productive if you adopt a professional grade tool such as *AtmelStudio* (page 2), *Visual Studio* (page 2), *Visual Micro* (page 2), or *PlatformIO* (page 2). And there will be times when you will want to switch to using one or the other on a given project. *Interoperability* means seamlessly switching from one tool to the other on the same set of files.
- *Chapter 6* - *Which IDE to work with?* (page 2) - We have all started with the Arduino IDE and are probably still using it for our daily Arduino work. But you would be considerably more productive, and your application would be more robust, if you were to adopt a professional grade tools such as *AtmelStudio* (page 2), *Visual Studio* (page 2), or *PlatformIO* (page 2).
- *Chapter 7* - *Arduino IDE* (page 2) - You probably know the Arduino IDE well. This chapter presents its strengths and weaknesses.
- *Chapter 8* - *AtmelStudio* (page 2) - If you are serious about programming Arduino, you owe it to yourself to look into how much more you can do with AtmelStudio as compared with the Arduino IDE, particularly when doing hardware-based debugging.
- *Chapter 9* - *Visual Studio* (page 2) with *Visual Micro* (page 2) is practically the same as AtmelStudio with Visual Micro. So, if you have switched from the Arduino IDE to AtmelStudio, why not do your programming with Visual Studio/Visual Micro, it is far more versatile and futureproofs your knowhow.
- *Chapter 10* - *Visual Micro* (page 2) - This plugin is available for both AtmelStudio (MicrochipStudio) and Visual Studio. It adds a serial monitor, an upload tool, serial-based debugging, and more. As for Visual Studio, it renders Microsoft's flagship IDE Arduino compatible.
- *Chapter 11* - *VS Code* (page 2) is a Microsoft foundation tool which is designed to be enhanced via extensions. PlatformIO, for example, is such an extension. It is well worth knowing how to use it as it could meet a diversity of needs (Python programming, *Perl*...).
- *Chapter 12* - *PlatformIO* (page 2), a VS Code extension, is a formidable tool which could

prove to be your choice if you intend to migrate to ESP32 or other microcontrollers.

- *Chapter 13* - *Other IDEs (Code::Blocks and MPLAB)* (page 3) - Code::Blocks is a good free IDE for C++ development but which, unfortunately, does not seem to be Arduino compatible. MPLAB is a professional grade tool for general Microchip microcontrollers embedded development.
- *Chapter 14* - *Debugging* (page 3) - Three debugging techniques are presented: print-based (*Print-based Debugging Framework* page 3), *Hardware-based debugging* (page 3), and *Serial debugging* (page 3) provided by Visual Micro.
- *Chapter 15* - *Good programming practices* (page 3) - Programming is a craft. A set of simple guidelines, easy to implement but which require a little discipline, should contribute to your improving your performance.
- *Chapter 16* - *Frameworks* (page 3) are application independent modules designed to accomplish specific tasks (data transfer, data storage, etc.). Taken together they constitute a toolbox which enhances reuse, robustness, and productivity.
- *Chapter 17* - Three *Should know tools* (page 3) are presented: *regular expressions*, *Awk*, and *Perl*. *Regular expressions (regex)* (page 3) is a tool which does text searches using wildcards. Do yourself a favor, learn the fundamentals of regular expressions. It will provide you with unsuspected possibilities when doing text analyses, such as search/replace, syntax analysis, etc. *Awk* (page 3) and *Perl* (page 3) are text processing programming languages designed to implement complete programs. *Awk* is basically used for search and replace. *Perl* is *Awk* with extended programming capabilities.
- *Chapter 18* - *From back of the envelope to final product* (page 3) - Once you have gotten the electronics to work, i.e. you have created a soldered prototype of your gizmo, you may have to do a lot more work such as creating a PCB, fabricating enclosures, sensor supports, and battery holders, whatnot. These require tools for creating PCB files (Gerber files with *Eagle* page 3), a CAD tool to create 3D parts (*Fusion 360* page 3), and get them 3D printed (Zortrax *3D printing* page 3). You will find descriptions of the tools I use and of other tools I did some research on.
- *Chapter 19* - The *Appendix* (page 3) expands on certain subjects: application code size, DOS box, memory use, frameworks, misdoings, *Awk*, and *Perl*.
- *Chapter 20* - The *Bibliography* (page 3) lists a few books which might be helpful to deepen your C++ know-how. It also includes Web links and YouTube videos.
- *A note on the book's source code* (page 3) - This book and its companion *Pragmatic C++ Arduino Programming* contain C++ code, C++ frameworks, and *Awk* and *Perl* programs. These are provided on an as is basis, free for noncommercial use, no guarantees whatsoever as to quality and suitability, under an MIT type Open-Source Licensing basis (go to *md-dsl.fr to download* the code). The initial release is incomplete, I shall improve on it progressively.
- *About the author* (page 3) - I, the author, have led a rich multi-national, multi-discipline life. My current interests are beekeeping, flying the airplane I built (RV8), creating electronic devices, 3D printing (Zortrax + Fusion 360), writing these books and possibly more to come, and much more a retiree indulges in.
- *Beehive weighing system* (page 3) - This two-book set, *Pragmatic C++ Arduino Programming* and *Defensive C++ Arduino Programming*, would not exist had I not embarked on creating

an *Arduino-based beehive weighing system*.

- The *Index table* (page 4) was designed so that you could find in it just about all of the book's content meant to be accessed by pertinent keywords.

In a nutshell, the contents of these two books cover C++ knowhow, good programming practices, IDEs and other tools, methodologies and procedures, and frameworks to help you be more productive when creating large, fast, compact, robust, and maintainable *Arduino-based* C++ applications.

Common sense and new tools

Developing the software to drive my *Arduino-based beehive weighing system* turned out to be an unanticipated adventure. I developed methodologies which I qualify as common sense and discovered new tools (AtmelStudio, Visual Studio, Visual Micro, VS Code, PlatformIO, *Perl*, *regular expressions*) which enabled me to be considerably more productive.

What follows is a set of common-sense directives and advice on tools to improve your programming and the application's quality.

- *Plan ahead* - If a module gets a little complex, step away from the computer for a spell. Sit down at a table with pencil, paper, and eraser to draw flow charts; list objectives, requirements, parameters. Brainstorm the module, how it works, what it is meant to accomplish. This phase is crucial. Assiduous attention to detail at this stage will save you tons of time later on. Furthermore, when coding directly on the computer, your hands being tied to the keyboards and your eyes glued to the screen, you are subject to tunnel vision and to getting lost in details. A large table with paper and books spread out liberates your mind. You are given free rein to enhance your creativity and imagination. You are freer to express itself. You will do a better job.
- *Top-down design, bottom-up details* - Use wrapper functions and C++ classes to define the big picture, *top-down* or *outside-in*. Use functions to handle details *bottom-up* or *inside-out*.
- Use a *smart C/C++ editor* - Imagine writing letters with *Notepad* as opposed to using *Word*. *Word* will format text, correct spelling, do grammatical error checks. The same holds with coding. A good programmer's editor will color text based on syntax, indent code, do code completion, correct and/or signal errors, and much more. It is important to use the correct tool for the job. *Notepad++* or the embedded editors in AtmelStudio, Visual Studio/Visual Micro, or PlatformIO are alternatives to the Arduino IDE.
- A good *C/C++ compiler* will generate good code and will error check or issue warnings. Fortunately, all four IDEs covered will do this (Arduino IDE, AtmelStudio, Visual Studio with Visual Micro, PlatformIO) since they incorporate the excellent GNU compiler.
- *Migrate to Visual Micro for Visual Studio* (page 5) - The Arduino IDE is great; without it, you most probably would never have adopted Arduino. Visual Studio/Visual Micro is more complete; the learning curve is short; and the Arduino sketch creation process is easy and works well. Your productivity and programming comfort will be immensely improved. They provide several other important features such as grammar checking as you type, `goto` implementation, direct from error reporting to source code, and more. Adopting them is a no brainer. What is more, it is possible to establish interactivity between IDEs, switch from one IDE to the another in less than 30 seconds, by activating an IDE specific `#define` macro. You could also envision using *PlatformIO* (page 6), but I find Visual Studio/Visual Micro more user friendly. Program with Visual Studio/Visual Micro; hardware-debug with AtmelStudio (see *Hardware-based debugging* page 6).
- Adopt *Good programming practices* (page 6) - There are resources (papers, videos, seminars, etc.) which advise on procedures and good programming practices (upper/lower-case, error checking, naming conventions, file organization). The sheer quantity is

overwhelming and generally addresses professional programming teams. Having looked at many of these, I compiled a subset of simple rules and procedures which should help you reduce errors and render programs more readable: Adopt naming conventions, break big functions up into several smaller functions thus easier to manage, comment your code, etc.

- *Frameworks* (page 6) - Source file organization, data storage, memory management, error reporting, etc. are tasks an application needs to implement. Frameworks respond to such needs; these ready-made tools enhance the programmer's productivity and *the application's robustness.*
- *Memory management* - Monitor RAM to avoid being out of heap or stack space. Bring things out of scope early to release memory. Optimize data storage. Whenever possible, encapsulate code in curly braces. Managing memory is one of the keys to robust, fast, compact, efficient code. See *Memory Management Framework* (page 6) and *Memory structure (appendix)* (page 6).
- *Documenting your code is often an afterthought*. My *Perl* based *Program Documentation Framework* (page 6) meets two objectives: document the code in a highly organized way (this cannot be overemphasized) and extract the corresponding documentation. The golden rule being *Comment! Comment! Comment!* Why not use a little discipline to structure these comments so that *Perl* could extract program documentation?
- *Use Perl* (page 6) *as a static checker of vicious bugs* - Simple mistakes like `=` instead of `==` in an `if` statement, or the reverse in an assignment statement, can remain unnoticed for ages. Attempting to discover a problem's origin can eat up precious time. A *Perl* program will extract all pertinent lines of code to enable you to peruse through them thereby detecting potential problems. I used it on my 15000 lines application and found three such mistakes. It saved me hours of debugging.
- *Regular expressions (regex)* (page 6) - If you are to use *Perl,* or profit from the fact that professional grade IDEs (AtmelStudio, Visual Studio, PlatformIO) search functions support regular expressions, then, by all means, learn how to use *regexes*. They can get cryptic however you only need a basic understanding to quickly undertake tasks you would have otherwise thought impossible. The learning curve is short.
- *Hardware-based debugging* (page 6) is a desirable activity since it should save on debugging time. Hardware-based debugging is based on interfacing the microcontroller with a hardware tool which enables step-by-step program execution (set breakpoints, display variable values). Atmel proposes many tools, 3 of which seem to respond to Arduino hardware-based debugging needs. These require that AtmelStudio be used.

  Serial debugging (page 6), provided by Visual Micro, is an additional debugging method. It lies halfway between print-based and hardware-based debugging. See *Debugging* (page 6).

Why choose Arduino

This book addresses the needs of Arduino developers who embark on major projects. By this I mean a project that contains subsystems which interact with each other: many modules, many conceptual entities, many files. The question remains: *Why choose Arduino for a particular project?* There are lots of other microcontrollers. *Why Arduino?*

Most of us started using Arduino because it is readily available, inexpensive, and particularly easy to work with. There are thousands of developers, possibly millions, which explains why there are libraries for just about every electronic component. A GitHub search on Arduino yields more than 100,000 results. And, it is inexpensive, but it carries a cost. Its main programming language is C++, a professional grade language which can lead the inexperienced to excessive debugging time. There is the alternative of using Python to program Arduino, but this carries important costs, greater memory requirements, slower application, and reduced programming possibilities. But programming is easier and faster.

It is a sure thing that, as with most things in life, a choice is a compromise. Certain criteria weigh more than others in a particular context thereby tipping the scales. Here are reasons for my having chosen Arduino and C++ for my *Arduino-based beehive weighing system* project:

- *Cost* - I developed my *Arduino-based beehive weighing system* for personal use, cost was not an issue as long as it remained reasonable. But, lurking in the back of my mind, if I succeeded in developing a sufficiently low-priced useful gadget, it could be turned into a money-making venture. Low component costs thereby became high priority. This implies the smallest possible program to minimize microcontroller size, thereby imposing C++ right away. All other development languages take up considerably more space and are slower. Hardware cost was another issue. My own PCB stand-alone based Atmel microcontroller would be less expensive than designs using board-based microcontrollers such as the ESP32 or RaspberryPi.
- *Development tools* - Most cross-development toolchains are costly from an Arduino developer's perspective. They are reserved for professional teams, one main provider being the Swedish company IAR Systems (check them out - *iar.com*); they propose a rich set of tools for just about every microcontroller. Luckily, Arduino developers may use one of several free development environments, the *Arduino IDE* (page 7) developed and maintained by the Arduino organization (*arduino.cc*), *AtmelStudio* (page 7), a Microsoft Visual Studio specifically designed for Arduino microcontrollers, *PlatformIO* (page 7), a plugin for Microsoft's VS Code targets embedded applications on numerous microcontrollers, including the ESP32. Finally, you may opt for *Visual Studio (Visual Micro)* (page 7), an almost identical twin of *Visual Micro for AtmelStudio (MicrochipStudio)* (page 7).

  Other IDEs (Code::Blocks and MPLAB) (page 8) are either not suited for Arduino development (Code::Blocks) or entail an extensive learning curve (MPLAB).
- *Sensors* - There are thousands of Arduino compatible sensors, boards, and devices. By Arduino compatible, I mean devices for which someone has developed an Arduino interface in C/C++. This alone can tilt the scale in favor of Arduino. There is no point in choosing a particular microcontroller if the desired device interface is not available.

- *Community* - When a problem arises, it is comforting to rely on a community of savvy developers who can respond knowledgeably to an issue. The Arduino community is huge. Just about every imaginable question has been posed.

Arduino with C++ was an obvious choice. I could have chosen RaspeberryPi because it is powerful, can be programmed in Python, and sits on top of an operating system, which makes it a minicomputer - keyboard, screen, and all. It is, however, more costly since it comes as a board as opposed to Arduino whose Atmel microcontrollers can be bought as stand-alone items.

The build toolchain

There is a lot more to getting a microcontroller to do what you intend it to do than downloading the Arduino IDE, creating a program, followed by a click on the build/upload button. This is what you would do when trying out the Arduino IDE with the *Blink* program. But, if you are to create an application which does substantial work, you will necessarily end up with many functions, many classes, i.e., many lines of code in many files. Upon attaining this level, you shall need to understand what is happening behind the scenes to remain productive, i.e., what the Arduino IDE is doing. You shall also need to look at debugging options other than putting occasional prints in your source code. And finally, you should examine professional grade IDE's; they will enhance your productivity and their learning curve is usually not steep.

This chapter covers the behind-the-scenes tools driven by the IDEs (compiler, linker, etc.). The chapters which follow cover the IDEs individually, starting with the Arduino IDE.

When you create a program with the Arduino IDE and click the build/upload button, a great deal of work gets undertaken behind the scenes. Understanding what is going on is essential to your being able to figure out what is happening when things get a little awry. The only tool you see is the Arduino C++ editor along with the IDE's toolbar. The behind-the-scenes tools (compiler, linker, etc.) take your C++ code and magically do what it takes for you to witness what your electronic gizmo does.

The build toolchain was extensively covered in the companion book *Pragmatic C++ Arduino Programming*. Here is a recap:

- *Intelligent C++ editor (next section)* - Creating a program relies on a good editor with which to write code. The source code is then passed to the preprocessor to handle macros, if any.
- *Make* (page 9) - C++ being multi-module, make handles which modules the IDE is to work on depending on their timestamp compared against the last build's timestamp. It controls work so that it should only process files which have been modified since the last build.
- *Preprocessor* (page 9) - C++ offers programmers a unique tool, the preprocessor. It enables doing text substitution, conditional inclusions, and file assembly. These tasks are controlled by macros which the preprocessor recognizes. Please note that this is an extremely useful C++ feature not available in most other mainstream languages although some may have third-party preprocessors available as plugins.
- *Compiler* (page 9) - Human readable programs (source code) need to be converted into machine code for the target microcontroller.
- The *Linker* (page 9) assembles modules and library components into an executable.
- The *Uploader* (page 10) loads the executable into the microprocessor so that it be stored inside one of its permanent memory, flash memory in Arduino's case.
- The *Bootloader* (page 10) is a program which resides in the microprocessor; it converts it into being its own ISP (In System Programmer). The ISP is electronic hardware whose role is to load a program into a microcontroller. The bootloader makes it so that the

microcontroller becomes its own ISP.

- *Serial terminal* (page 10) - If you want to send information to the outside word, you can do so by displaying it on a screen connected to the microcontroller via a serial port. Your application might not require your doing so; but, during the development phase, you might want to investigate what it is doing. Your only recourse might be to print values as the application runs; this is when a serial terminal comes in handy.
- *Hardware setup* (page 10) - Developing your gizmo may start out by piecemeal breadboard proof-of-concept work gradually evolving to a final SMT (surface mount technology). There are different types of hardware you may use at each stage of the development process (breadboard, prototype board...).
- *Debugging* (page 10) - It is a fact of life that as you program, you make mistakes; your program fails to work properly - it has bugs. Finding bugs is what debugging is all about. You can do so by placing prints to get to know what your program is doing; or, you can use a hardware-based debugger to stop the application as it runs, and look at the program's state.

These are summarily described below. *Debugging* (page 10) is described in its own chapter.

1.1 Intelligent C++ editor

If you are to create a C++ application for your gizmo, you must write the code into a file and feed it to the build toolchain. The editor should do a lot more than just indent code according to its scope level. It should color code according to syntax; do name completion to help you avoid misspelling already defined variables; functions, etc.; detect programming errors from improperly written constructs, (missing parentheses, missing items,...); report undefined entities...

Dedicated programmer editors such as Notepad++ and Arduino IDE V1.8 are basic editors. Arduino IDE V2 begins to provide some editing enhancements. *AtmelStudio* (page 10), *Visual Studio* (page 10), and *PlatformIO* (page 10) incorporate intelligent editors which vastly improve programmer productivity.

1.2 Make

If your program is small, one or two small modules, you may resort to recompiling everything and create the executable. But what if your program is large, 40 modules, 100,000 lines of code. You should only recompile modules which have been modified since the last build, one or two instead of forty. The IDE should create a make file which establishes which modules to recompile and which modules to assemble to create an executable. Fortunately, all IDEs covered in this book use *make* to control the build process.

1.3 Preprocessor

C++ has a unique feature referred to as macros. These are special variables which begin with a # (pound sign) which the preprocessor recognizes. They enable accomplishing three fundamental tasks:

- *Text substitution* - If you write `#define BUFFER_SIZE 256`, the preprocessor replaces every occurrence of `BUFFER_SIZE` with `256`.

- *Conditional inclusions* - You may want to differentiate targeting an ATmega328P from an ATmega2560. You can do this by placing code inside an conditional inclusion such as `#ifdef ATMEGA328P...#else...#endif`.
- *File assembly* - Files can be pasted inside other files via a macro such as `#include <Arduino.h>`.

C++ is unique in that no other language except possibly Fortran formally offers this feature. You may however find third-party preprocessors for other programming languages (Python in particular).

1.4 Compiler

The compiler transforms human readable source code (*.h/.cpp* file) into machine code (*.o* file). Because C++ applications can be very large, they are multi-module. The compiler works on one module at a time.

1.5 Linker

The linker creates your program. Once the compiler has finished compiling all of a program's modules, the linker assembles them, along with library components, into a whole, an executable, also referred to as image (usually a *.hex* file).

1.6 Uploader

Once the executable has been created, it must be uploaded into the microcontroller. A specialized program, *avrdude*, handles this task. It sends the program to an ISP, electronic hardware whose job is to load (burn) the program into the microcontroller. One of the neat features of Arduino boards resides in their becoming their own ISP via installed software, referred to as a *bootloader* (next paragraph).

1.7 Bootloader

The ISP (In System Programmer) is a piece of electronic hardware which receives the program via its com port and transfers it into the microcontroller's memory. An ISP must consequently be placed between the development computer and the microcontroller board. The *bootloader* is a program installed in an Arduino board which enables it to become its own ISP; an external ISP is no longer needed. The price to pay is little less flash memory.

1.8 Serial terminal

It is all good and well to create a gizmo which opens a chicken coop door at sunrise and closes it at sunset. But, during the development process, you may want to display info to understand what the program is doing. You do this by writing to one of your computer's serial terminal applications. The terminal could also be a LED screen connected to your board.

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*.

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 39). It began as a back of the envelope 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 and later to Visual Studio/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 the literature and the Internet were lacking in resources for amateur programmers who already programmed in C++. Or rather, there was 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.

Beehive weighing system

Life being a collection of chance encounters, a friend of mine (Charlie H.) introduced me to beekeeping, an immensely rewarding activity. One of my first tasks as a new beekeeper was to monitor beehive weights via a hand-held scale. This information is crucial to assess how bees are doing.

Another chance encounter (Daniel T.) introduced me to Arduino. It seems that life is conditioned by chance encounters. Anyway, as soon as I understood what I could do with Arduino, I asked myself: *Why not hack an electronic Arduino-based beehive weighing system?* On the surface the idea was simple enough: get strain gages cannibalized from ordinary bathroom scales, throw in opAmps, voltage regulators and a few other little gizmos. Add to that a GSM board, a radio, and a clock and I would get the beehives' weights, temperatures, humidity, and battery voltages on my cell phone. Simple! Nothing to it! Well, not quite.

To make a long story short: *beekeeping + Arduino + C++ = electronic Arduino-based beehive weighing system = challenge*. Little did I know how much I did not know and just how much work would be required to bring this project to fruition. Having defined how to get the data from the beehives, and how to get them into my cell phone, I trudged along, slowly, and painfully. I grossly underestimated the work and amount of learning required to get the system to function as I would like it to.

My electronic know-how being limited to Ohm's law, the challenge proved to be daunting. Being retired, with nothing in particular to do aside from keeping fit, maintaining social ties, and being a law-abiding citizen, this project quickly became a full-time job. I was never bored. I had to crash course transistors, opAmps and voltage regulators. Discovering this world was fun. Getting an NPN/PNP transistor-based latch to work, and understanding why it worked, brought me intense satisfaction.

As for Arduino, what a discovery! The microcontroller and the Arduino IDE are fantastic and cost next to nothing. My enthusiasm was immediate. I first got breadboard proof-of-concept prototypes working. I got strain gages which I pulled out of an electronic bathroom scale to deliver a signal via an opAmp. Radio communications were based on XBee (Zigbee protocol). I chose the DS3231 RTC clock to synchronize the units to minimize wakeup time. An A6 GSM board established two-way SMS communications with my cell phone.

1.9 Beehive weighing system architecture

As I became savvier with the electronic components I could use, system characteristics slowly percolated out of a host of possibilities:

- *Parent-child architecture* - I opted for a parent/child star network system as opposed to independent beehive modules, each reporting directly. Children would be on beehives. They generate data and send it to the parent (coordinator) which would be near-by. It would collate data from the beehives and send the info to my smartphone.
- *RF communications* - The coordinator and beehives communicate with each other via RF communications, no wires between the beehives. I chose a ZigBee based system (XBee).
- *Batteries only* - Since beehives are in remote locations, they require electrical autonomy, making batteries necessary (18650 LiIon). I rejected the use of solar panels since they

would be conspicuous, obviously valuable items. There was every likelihood that they get stolen.

- *Cost* - If the system were to become a commercially viable product, cost would be an issue.
- *Ease of use* - The system needed to be flexible: When to weigh and how often, get data to trigger notifications on specific events, etc.
- *Interface with the outside world* - I had to decide how to interface with the system (Firefox, Lora, WiFi, Bluetooth, cell phone). I chose an A6 GSM board to communicate with my cell phone as it seemed to be the easiest, most practical, and immediate solution.

Given three years of hindsight, I must say that the electronic aspect of the project was indeed easy. However, the C++ program to manage the system proved to be a lot more complex than anticipated. I was no longer inside an *.ino* sketch mindset - direct programming from mind to computer. The application grew inexorably to 35 files of code at last count. I had to improve my C++ know-how, develop new techniques, and change my mindset - professionalize my work.

As for my know-how, I had C++ experience. I poured over Kernighan and Ritchie's *The C programming Language*. I then went on to Bjarne Stroustrup's *The C++ Programming Language.* But my experience was rusty. I still have early editions of these books; I consider them collectors' items.

I gradually improved and organized my programming by various means: C/C++ know-how, good programming practices, frameworks, and better IDEs (AtmelStudio, Visual Studio + Visual Micro, PlatformIO). These helped me attain the desired objectives: productivity, application maintainability and robustness, speed and compactness.

I have finished the first version of my *Arduino-based beehive weighing system*. The next phase is to run it through its paces on several beehives and establish its resistance to weather. After that, if I were to make it into a commercially viable product, I would need to undertake fairly extensive work, incorporate less expensive components, use a cheaper radio, create alternative interfaces to the outside world, and use SMT based PCBs to lower costs.

Where to from now

The current version of the beehive is functional however, several slight problems remain:

- The XBee radio modules are far too expensive ($30.00 apiece). Current work centers around using BRF24L01modules. They come in at around less than $3 apiece.
- Using a cell phone to get data is cumbersome because of the limitations of an SMS message and costly because of the requirement to pay for SIM cards.
- A friendly Internet based user interface to visualize data and schedule jobs needs to be put together.
- Envision migrating from through-hole to surface mount technology (SMT).

As soon as these books get published, I shall resume work on the *beehive weighing systems*.

Index table

Symbols

C

E

Q

R

W

DEFENSIVE C++ ARDUINO PROGRAMMING

The right tools for Arduino C++ programming

If you are you are constantly frustrated due to fighting against the tools you are working with, or feel that you are reinventing the wheel, this book is for you.

Why Arduino and C++ are an excellent choice for embedded software development.

Good programming practices help getting it right the first time around.

Which IDE should you work with? Arduino, AtmelStudio (MicrochipStudio), Visual Studio plus Visual Micro, Microsoft VS Code, PlatformIO, Code::Blocks, MPlab?

Visual Studio plus Visual Micro is, IMHO, the best tool to develop Arduino applications with. This book will guide you on using this tandem and details the process of importing Arduino sketches. The learning curve is short.

Print and hardware-based debugging are covered extensively. These are good reasons for you to develop with Visual Studio plus Visual Micro and use AtmelStudio to hardware debug with ATmega XMini boards.

The new Arduino IDE version 2 is presented. Is it ready to replace version 1.8.19?

Understand what lies under the IDEs' hoods. Behind the scenes, the IDEs drive the GNU toolchain (*preprocessor, compiler, linker, make*) - *avrdude* to upload code into your microcontroller.

Seventeen C++ frameworks save you from inadvertently reinventing the wheel. Full source code available in *https://md-dsl.fr*.

C++'s exception handling not being supported by the Arduino environment, use the Pseudo Exception Handling Framework, `setjmp/longjmp` based.

Awk, *Perl*, and *regular expressions* enable you to undertake tasks otherwise not feasible. A *Perl* program detects virtually undetectable silly mistakes (download link in *https://md-dsl.fr*).

Manage memory to determine its status (*contiguous* and *fragmented memory*). Anticipate *memory requirements* and discover *memory hogs* and *memory leaks*.

The companion book, *Pragmatic C++ Arduino Programming*, explains C/C++ fundamental mechanisms, the preprocessor, the PROGMEM framework, the many gotchas C/C++ can throw at you, how to manage memory, and a lot more.

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

Being defensive as you write code means being aware that your path is paved with danger. Discover how nasty C/C++ gotchas can get and how using professional grade tools can save you time and take the hassle out of developing with C/C++.

9 782958 562823

Kindle $3.85 Paperback $29.85 Hardcover $34.85

Warthog drawing by the author