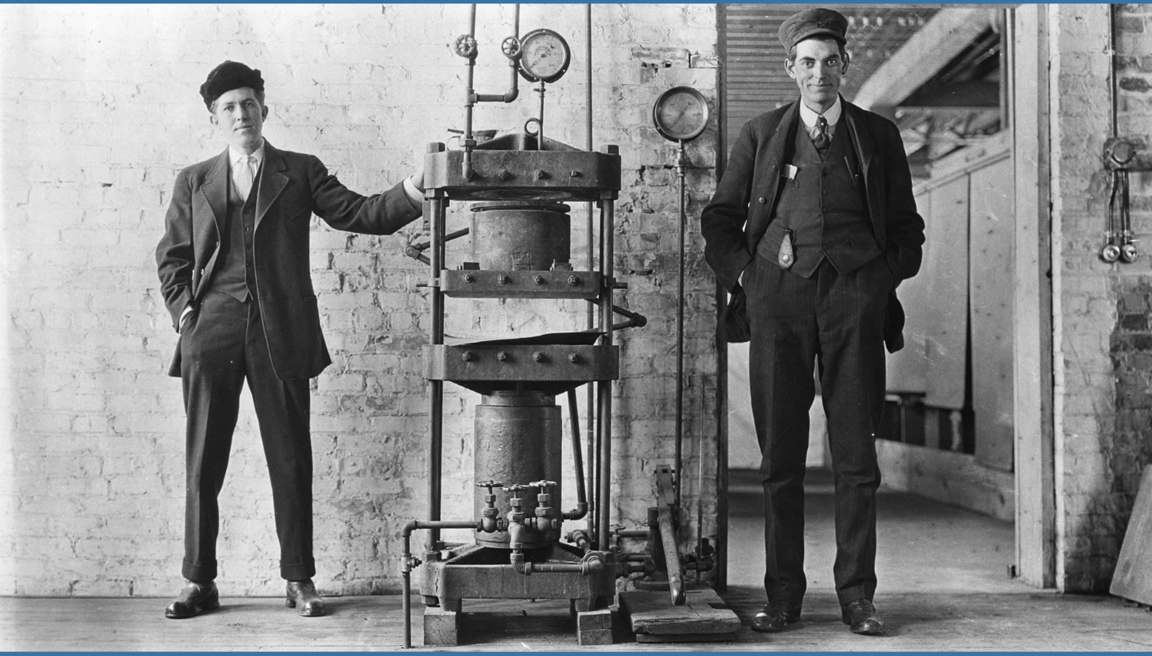
O'REILLY®

The Resurgence of C Programming

Do You Still Need to Write Code to Build Cool Machines?



Mike Barlow

The Resurgence of C Programming

Do You Still Need to Write Code to Build Cool Machines?

Mike Barlow

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

The Resurgence of C Programming
by Mike Barlow

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editors: Brian Jepson and Susan Conant
Production Editor: Nicholas Adams
Copyeditor: Rachel Monaghan
Interior Designer: David Futato
Cover Designer: Randy Comer

December 2016: First Edition

Revision History for the First Edition
2016-12-09: First Release

978-1-491-93280-3

[LSI]

Table of Contents

The Resurgence of C Programming

Way back in the early 1970s, learning C was a rite of passage for many students. By today's standards, it's not a very high-level language. But back in those early days, long before the arrival of Java and Python, C was considered high level, especially when compared to assembly languages.

In the preface to their book *The C Programming Language*, Brian W. Kernighan and Dennis M. Ritchie note that C "is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages."

To some degree, C was written for the purpose of elevating UNIX from a machine-level operating system to something resembling a universal platform for a wide range of software applications. Since its inception in 1972, C has been the common language of UNIX, which essentially means that it's everywhere.

Example 1-1. C program to blink an LED on an 8-bit microcontroller with a 2,000-microsecond delay (adapted from the book AVR Programming)

```
/* Blinker Demo */
// ------- Preamble -------- //
#include <avr/io.h>/* Defines pins, ports, etc */
#include <util/delay.h> /* Functions to waste time */

int main(void) {

  // -------- Inits --------- //
  DDRB = 0b00000001; /* Data Direction Register B:
                        writing a one to the bit
```

```
                                     enables output. */

  // ------ Event loop ------ //
  while (1) {
    PORTB = 0b00000001; /* Turn on first LED
                           bit/pin in PORTB */
    _delay_ms(2000);    /* wait */
    PORTB = 0b00000000; /* Turn off all B pins,
                           including LED */
    _delay_ms(2000);    /* wait */
} /* End event loop */
return (0); /* This line is never reached */
}
```

C and C++ are at the heart of Arduino, the open source project for building do-it-yourself devices and hardware. "Arduino code is essentially C and C++," says Massimo Banzi, a cofounder of the Arduino project. "Right now, you can write Arduino code on an 8-bit microcontroller and then on an ARM processor. You can go right up to a Samsung Artik, which is essentially a Linux machine with an 8-core processor. We can run Arduino on top of Windows 10."

Example 1-2. The same behavior as Example 1-1, using Arduino's simplified C dialect (from the book Arduino Cookbook, 2nd Edition)

```
const int ledPin = 13; // LED connected to digital pin 13

void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  digitalWrite(ledPin, HIGH); // set the LED on
  delay(2000); // wait for two seconds
  digitalWrite(ledPin, LOW); // set the LED off
  delay(2000); // wait for two seconds
}
```

Learning About Software by Tinkering with Hardware

How does this play out in the real world? Let's say you're an aerospace engineer and you're asked to improve the functionality of an actuator that moves a control surface, such as an aileron on the wing of an airplane. You might begin by using components from an Arduino kit to create a low-cost prototype of the actuator.

After you've got your prototype working, you can tinker around with it and optimize its performance. "If you are an engineer, you can take your idea and use Arduino to build prototypes very fast," says Banzi. "So you might begin with Arduino and then decide to reimplement the code using another tool. But you might also wind up using Arduino all the way. Or you could use a C compiler to move your code to a piece of hardware that doesn't run Arduino, but will run C or C++."

Transferring the code isn't difficult. The hard part, says Banzi, is writing the algorithm that controls the actuator moving the aileron. Fixing the problem with the aileron means you need to develop a new algorithm. Then you need to tweak and adjust your algorithm. "If it were my project, I would write my Arduino code for the tool and then I would use a pure C or C++ file for my algorithm," says Banzi. "Once my project is working and I can tweak the algorithm, I would just take the algorithm and paste it into the tool."

If you're working in closely regulated industries—such as automotive, aviation, or healthcare—you might need to redesign your Arduino prototype before deploying it. But there are lots of situations in which regulatory compliance wouldn't pose a hurdle.

Let's say you decide to build a digital watch for yourself. You could buy a Time II DIY watch kit from SpikenzieLabs, follow the instructions that come with the kit, and create a nifty timepiece. But here's the really cool part: the watch is designed to be hackable. You can reprogram the watch with Arduino IDE (integrated development environment) software. Basically, the kit empowers you to create a one-of-a-kind smartwatch by tweaking a few lines of Arduino code.

Working in Tight Spaces

Some obstacles are more difficult to overcome than others. When you're building an open source wristwatch, for example, you might decide to follow your muse, ignore the instructions, and install a bigger battery or use a different color display.

"If you're designing it yourself, you're going to have a lot more research to do because you're going to need to know how much power you want to use. You're going to need to figure out how to program the display and what type of display you want to use," says Brian Jepson, an author, editor, and experienced digital fabricator. "Once you've got the parts, you've got to put them together and pack them into a case that you can wear on your wrist."

Manufacturing a custom case for your open source watch will likely involve 3D printing, which often requires some basic programming skills. "Each part has to sit really close to the other parts. You don't have a lot of space. You can't have wires running too long or too short. It's a tight fit and you have to make sure you make good, reliable connections between the components," Jepson says.

When all the parts are connected and packed securely in place within the case, you can seal it up. But your work isn't necessarily done at that point. "If I were building it from scratch, following somebody else's instructions, I might have had to program it myself. Or, if I ever want to customize it, to display things in a different way, maybe to graph things, I'd have to download the source code, make some changes, and then load it onto the device," says Jepson.

Unforeseen Consequences

In a cosmic sense, Arduino fulfills the vision of C's creators, who foresaw a world of portable software applications. What they did not anticipate, however, was the emergence of open source hardware, the maker movement, and maker culture.

For the early pioneers of computing, creating functional software was an end in itself. Today, it's not enough to just create software that runs without crashing. The world wants software that can run devices and machines such as trains, planes, automobiles, and pacemakers—without crashing.

That puts a lot of weight on C, which is used widely to write embedded software. "C is the Latin of programming languages," says Ptah Pirate Dunbar, an open source hacker and professor of computer science. He notes that many commonly used high-level languages are influenced by C through syntax, function, or both. "Learning C empowers developers with the mental flexibility required for transitioning across C-influenced languages with ease and agility."

George Alexandrou, vice president and chief information officer at Mana Products, a global manufacturer of private-label cosmetics, says there are two kinds of software developers: mechanics and engineers. “The mechanics will fix things when they break. But when you want to design something new, you need an engineer,” says Alexandrou. “If you want to be an engineer, you need to know how to program in C. Programming in C can get you out of trouble. It can also get you into trouble.”

Extracting Maximum Performance

Suman Jana, an assistant professor in the Department of Computer Science at Columbia University and a member of the Data Science Institute, says the C language “allows expert programmers to extract maximum performance from the underlying hardware resources.”

With C, programmers can control and customize almost every aspect of their programs. “However, as a side effect, it is also very easy for a programmer using C to inadvertently make serious mistakes, like memory corruption, that lead to security vulnerabilities,” Jana says.

Buffer overflow is a “classic example of memory corruption in C code,” he says. It can happen when a programmer copies data into a preallocated buffer without checking whether or not the data will fit into the buffer. If the user input is longer than the buffer, it will overwrite past the boundary of the buffer. That can pose serious security risks. When user input, for example, is written into a buffer without bounds checking, the system can be susceptible to an injection attack. Why is that? When buffer overflow is triggered by user input, the user can probe the system and potentially take control of it.

Caveats aside, C is especially relevant to developers working on embedded software for smart machines. “Embedded software often runs in resource-constrained environments. The target devices have small memory and limited computational power. For such environments, C is a very good fit because it allows programmers to get good performance even with limited resources,” says Jana.

“Learning C is a good exercise to understand the underlying system and hardware. Even if a Java/Python programmer does not use C on a day-to-day basis, learning C can potentially help them understand how different Java/Python features are actually implemented by virtual machines. In fact, many Java/Python virtual machines are written in C.”

Jana sees plenty of good reasons for coders to learn C. "The C community is a large and thriving group. Some of the most popular and widely used software—like the Linux kernel, Apache HTTP server, and Google Chrome browser—are written in C/C++," he notes.

Still Alive and Well

While there might be some vague similarities between C and Latin, there are also stark differences. Latin is a dead language. C is most definitely alive and well. A hardware hacker recently compared C to Jason Voorhees, the main character in the Friday the 13th series of horror movies. Just when you think Jason is dead, he comes roaring back to life.

"Hardware has gotten more complex and there's more to debug," says John Allred, an experienced developer of embedded software and hardware interfaces. "Nowadays, the programmer is expected to help with the debugging. I still believe that C programmers have a mindset that helps them see the big picture. When you know C, it helps you solve problems with hardware. It gives you a different way of looking at the world."

Allred is senior manager of cybersecurity at Ernst and Young (EY). After graduating from MIT and before joining EY, he participated in a number of technical firsts: the first internal hard drive for the Macintosh; SIMNET, the first large-scale networked simulation of military vehicles; and RTIME, the first large-scale networking engine for video games and distributed systems. He is an unabashed fan of older programming languages like C, which require a thorough understanding of how computers actually work.

"When you program in C, you control the memory. But when you program in Java, it does the memory management for you. When Java decides it's time to clean up the memory, it goes ahead and does it—even if you're in the middle of doing something else," says Allred. "With Java, there's a higher level of abstraction. You're not driving the hardware directly."

If you're writing code for an online retailer, the loss of 100 milliseconds probably won't ring any alarm bells. But when you're programming software for real-time applications, lost moments can make a big difference. "When you're writing code for drones or driverless cars or oil refineries—situations where you need real-time performance—then Java and Python shouldn't be your choices," says Allred.

The Right Tool for the Job

In some respects, Allred represents a vanishing generation of programmers who are comfortable with assembly code and have a deep appreciation for its intrinsic value. Bare-metal coders are in the minority, but many of their ideas are gaining new currency as the lines between software development and hardware design become less distinct. "I think it's always really good to know how the software interfaces with the hardware," says Limor "Ladyada" Fried, founder of Adafruit. "Understanding the limits of the hardware will help you understand optimizations that are possible in the software."

If your project involves pushing data from a sensor across a wireless link, for example, you need to know the limitations of the hardware. Until you actually begin experimenting with that wireless link, says Fried, you won't know how much data it can handle. In those types of situations, which are becoming more common, there is no substitute for hands-on experience.

Ideally, your choice of a programming style should be determined by the requirements of the project before you. "I think it depends on your needs," says Fried. "There are some times when you're really optimizing and you want to go to machine code or FPGAs [field-programmable gate arrays] or CPLDs [complex programmable logic devices]; especially when you're doing extremely advanced, very time-sensitive stuff like SDR [software-defined radio] or video extreme handling, you might go with an FPGA. For most people, C or C++ seems to dominate."

Simon Monk, a prolific author and experienced builder of open hardware projects, says C is still the optimal choice for programming on machines. "On a microcontroller, I would always use C. It's a good compromise between the performance of assembler and the readability of a high-level language," Monk says.

But he is not enthusiastic about the idea of hardware developers replacing software engineers. "With a few honorable exceptions, hardware developers should not be allowed to program…although they would probably say the converse is true of software engineers like me designing electronics," he says.

He faults machine code for being “almost universally opaque and badly structured, with functions dozens of lines long.” And he raises an interesting point about the inherent differences between hardware and software developers, arguing that hardware developers are often more focused on how a particular piece of hardware works, instead of thinking more deeply about the service the hardware is designed to provide.

For software engineers who want to write embedded code for hardware, Monk advises taking small steps. “Don’t use the first library you find on GitHub. Try out a few examples and if you don’t like the API or the code smells, don’t be afraid to write your own,” he says. “Also, don’t get carried away with overstructuring or overpatterning your code. A 50-line Arduino program does not benefit from being split into three classes and a suite of unit tests and an implementation of the observer pattern.”

“It’s Like Learning to Drive a Stick Shift”

Edward Amoroso, chief executive officer of TAG Cyber and former chief information security officer at AT&T, says knowing C is handy, but no longer absolutely essential. “You can drive your car to Buffalo and not know how the engine works,” says Amoroso. “I think the analogy holds for software. On the other hand, if something goes wrong or some kind of weird issue arises and you have no understanding of the underlying logic of the software and the hardware, it might be more difficult for you to fix the problem.”

From Amoroso’s perspective, the advantages of knowing C are at best marginal in today’s world of highly abstracted software. “It gave you a huge advantage 20 years ago and a good advantage 5 years ago,” he says. “In the near future, however, it will probably be even less of an advantage, given the level of abstraction and the power of translators.”

Still, he isn’t ready to throw in the towel and abandon C to the ash heap of history. “If you’re programming on bare metal, then it’s helpful to know C. It’s like learning how to drive a stick shift—it gives you more control,” says Amoroso. “But for the most part, the trend in software development is more toward the logical connection of working software components that are plucked from libraries and put together.”

Amoroso says he misses the old days when students would actually learn to *write* code. "Today, much to my chagrin, young people are taught to code using programming environments where they're moving widgets around and creating little games. They're basically adding logic to widgets. I'm not saying it's necessarily harmful, but I'd rather see them learn how a computer operates first, and then build up to writing software. But that's not the way it's typically done in schools today."

Learn by Doing

Educators and cognitive psychologists have long known the value of hands-on learning. When you work with your hands, your brain is actually more engaged than when you are simply listening or reading.

Dale Dougherty, the founder and CEO of Maker Media, sees a renaissance in hands-on education. Nowhere is this renaissance more evident than at Maker Faire, a series of live events produced by Maker Media. Maker Faire is "a family-friendly festival of invention, creativity, and resourcefulness, and a celebration of the maker movement."

At a recent Maker Faire at the New York Hall of Science, Dougherty emphasized the importance of combining "art, science, craft, and engineering" in bold new ways that encourage both creativity and technical capabilities. "What we see here at Maker Faire is a flourishing of creative spirit and technical skills…coming together to make new things possible," he said in a talk at the event.

Visitors to the event saw dozens of exhibits from makers of drones, robots, 3D printers, electric vehicles, rockets, wearables, alternative energy solutions, and various combinations of hardware and software.

Dougherty is one of several voices in the maker movement, which has already transformed or disrupted many aspects of traditional technology culture. The Raspberry Pi, for example, was developed by a team at the University of Cambridge's Computer Laboratory as a tactic for enticing more students to study computer science.

The first version of the Pi, which is essentially an inexpensive single-board computer, was released in early 2012. Since then, the Raspberry Pi Foundation has sold 10 million devices.

Although the Pi was initially designed for students, it is now used widely by engineers and developers all over the world—a truly amazing demonstration of the maker movement's influence, both inside and outside the classroom.

Everyday People Making New Tech

Tom Igoe is an associate arts professor at ITP, a two-year graduate program within the Tisch School of the Arts at New York University. Officially, ITP's mission is exploring "the imaginative use of communications technologies," and it's become a launch pad for the expression of creativity through novel combinations of hardware and software.

Igoe leads two areas of curriculum at ITP: physical computing and networks. He has a background in theater lighting design and is a cofounder of the Arduino open source microcontroller environment.

"All of the various technologies we're talking about play a big role in our everyday life. It doesn't matter whether we are a technologist or engineer or whatever, they influence our everyday life," says Igoe. "But if we don't have an understanding of them…then we don't have much control."

One of the ITP's primary goals, says Igoe, is "breaking the barrier" that prevents or discourages people from making their own devices and machines. One of his students was a physical therapist who wanted to create a device for helping patients improve their range of motion following an illness or injury. "She didn't just want to describe what she wanted and have someone else build it," Igoe explains. "She wanted to build it herself, so she could make the design decisions based on her actual experience with her patients."

From Igoe's viewpoint, the traditional tech industry has become too doctrinaire in its approach to valuing talent. "They assume that technical proficiency is the only measure of a person's work. The truth is that you need people with a lot of different capabilities," he says. "You need a lot of different skills and a lot of different ways of seeing the world."

It's not necessary for every student to become an engineer or designer. "When I've got students who are not technically gifted, I try to help them figure out what they're really good at and apply it to making the kinds of things we're talking about," says Igoe.

Much of the inspiration for Arduino, says Igoe, came from working with students who were not engineers, but who were very interested in using electronics and controllers in their projects. "Arduino is a microcontroller that's not designed for engineers; it's designed for everyday people," he explains.

That said, Arduino is now used by a wide variety of people—including engineers and designers—as a physical sketchbook for hardware projects. Increasingly, it's seen as an integral part of the prototyping process.

The popularity of low-cost kits based on Arduino or Raspberry Pi components has led to "an explosion of creativity," says Mark Gibbs of Gibbs Universal. He is an author, journalist, and serial tech entrepreneur. "It's easier now for people to enter the tech market with little or no technical knowledge and then acquire the skills they need to become very proficient at building devices and machines."

It's not just the low cost that makes Arduino and Raspberry Pi so attractive—they're also easy to work with, says Gibbs. "The impact has been enormous. Now you have the ability to bring technology into the arts, which is a huge thing in itself because it changes the nature of what we consider to be art."

Gibbs cites the example of Sketchy, a drawing device created by Richard Sewell (aka Jarkman) that combines an Android phone with a delta robot based on Arduino components. Another example is Robot Army, a team of artists and engineers that make easy-to-build robot kits for people interested in learning and experimenting with inexpensive robots. "Those kinds of projects would have been inconceivable just a few years ago. You would have needed hundreds of thousands of dollars to build a robot. Now you can build one for under $100," says Gibbs.

Blurring the Boundaries

The professional engineering community is not immune to the lure of low-cost, easy-to-use tech like Arduino and Raspberry Pi. "In the beginning, many professional developers laughed at our tools," recalls Banzi. "They thought they were toys or kind of stupid."

Many of those professionals are no longer laughing. “Now they’re using our tools to build quick and inexpensive prototypes,” says Banzi. “More important, developer teams use tools like Arduino to onboard beginner programmers more quickly. Now you can give new programmers small tasks and projects that will help them build their skills faster.”

In many ways, the maker movement has blurred—or in some cases, erased—the traditional boundaries between professional and amateur science. While some people might reflexively argue against that type of blurring, it harkens back to the times before the Industrial Revolution, when practically all scientists were amateurs. Today, it seems we no longer have to choose between being amateurs or professionals—we can enjoy the best of both worlds, whether we choose to write code in C or assemble parts from a kit.

About the Author

Mike Barlow is an award-winning journalist, author, and communications strategy consultant. Since launching his own firm, Cumulus Partners, he has worked with various organizations in numerous industries.

Barlow is the author of *Learning to Love Data Science* (O'Reilly, 2015). He is the coauthor of *The Executive's Guide to Enterprise Social Media Strategy* (Wiley, 2011) and *Partnering with the CIO* (Wiley, 2007). He is also the writer of many articles, reports, and white papers on numerous topics including smart cities, ambient computing, predictive maintenance, advanced data analytics, and infrastructure.

Over the course of a long career, Barlow was a reporter and editor at several respected suburban daily newspapers, including the *Journal News* and the *Stamford Advocate*. His feature stories and columns appeared regularly in the *Los Angeles Times*, *Chicago Tribune*, *Miami Herald*, *Newsday*, and other major US dailies. He has also written extensively for O'Reilly Media.

A graduate of Hamilton College, he is a licensed private pilot, avid reader, and enthusiastic ice hockey fan.