

Why C & C++? A GATS Companion.

Author: Garth Santor (with Trinh Hân)

Copyright Dates: 2020, 2021, 2023

Version: 1.2.0 (2023-12-28)

Overview

Unfortunately, the case for studying C and C++ is not as obvious as it once was. Social media, while giving voice to many of the issues that face these languages, has also driven many falsehoods into increasingly popular belief.

In this document I will present my arguments for contemporary programmers and students to learn C and C++.

Introduction

Why bother? C is around 50 years old, and C++ is almost 40. Computers today are nothing like the machines that C and C++ were designed for. Shouldn't we use more modern languages? C and C++ give programmers nightmares! Why are we still using them?

My father (an historian) always told me that you cannot begin to discuss the merits of a thing without knowing its history. This is particularly true of language. The very words we create are shaped from our history. For example: the term *bug* used to describe a fault in a computer was literally a bug (it was a moth). In 1947, a team of computer engineers working at Harvard University found that a moth had been trapped in Relay #70 Panel F and was the cause of the machine's failure. Admiral Grace Hopper taped the "bug" into her notebook (now at the Smithsonian Institute). While she never used the term "debugging" in their logs, the story spread and the terms "bug" and "debugging" are now the most widely used jargon to describe failures in, and the activities to repair computers and its software.

Languages come and go. At any give time in the last thirty years, over 2500 languages can be counted to be in use, with over a hundred new languages being created each year, with a similar number of languages falling into disuse. So, which is the 'best' to learn? Python is a popular introduction language being studied in elementary schools, Java is conventional, Go and Swift are cool, and COBOL just won't die.

Is there one language that is a better springboard to the other languages? Can you say with confidence, "I should have no problem learning COBOL, because I know JavaScript?" But most people do believe us when we say, "I should have no problem learning Java, because I know C++."

We believe C & C++ to be the fundamental language for professional computer education. Within C & C++ a student of computing will learn the language that virtually all operating systems have been written in. C & C++'s low-level capabilities will help them understand how software works at a fundamental level. Their high-level capabilities will expose them to the major paradigms of computing. We believe that a student of C & C++ can move to more languages with less effort than from any other starting point, and that C & C++ will make a better programmer regardless of the language(s) you ultimately find a career programming.

To make our case we will examine the following topics:

1. Philosophical perspective: what type of languages they are (ontologically), the languages' stated purpose (teleology) and by its apparent development philosophy.
2. Influence on other languages.
3. A brief examination of benchmarking, popularity, and a survey of their application.
4. A review of criticisms, and a critique of the criticisms.
5. Alternatives to C & C++.
6. A rubric for when to use these languages.

7. A discussion of why to learn C & C++.
8. The future of C & C++.

History of C and C++

C and C++ are oddly entwined languages. Some consider them to be two separate languages, while others – a single language. We believe that C++ is a separate language that both respects its heritage as the premier C-derived language and builds upon that language to bring the programming paradigms and features that C doesn't possess. C in turn poaches from C++ those features that it can digest without betraying its paradigm as a purely procedural language. They have separate standards committees, yet they work together and are influenced by each other to a greater degree than any other two languages.

[1960s] before “C”

In the 1960s, *application software* was typically written in a 3rd generation language such as Fortran or COBOL, whereas *system software* was written in 2nd generation assembly languages. Let's get a sense of those differences.

COBOL

COBOL, the “COmmon Business-Oriented Language” was created by the US Department of Defense as a *portable* programming language for data processing. The target audience was not computer scientists or engineers, but accountants and other business professionals.

Let's look at a simple COBOL program.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUMSERIES.

DATA DIVISION.
WORKING-STORAGE SECTION.
77 N    PIC 999.
77 I    PIC 999.
77 Sum  PIC 9(20).

PROCEDURE DIVISION.
DISPLAY "Enter the limit: ".
ACCEPT N.
PERFORM Series-Loop VARYING I FROM 0 BY 1 UNTIL (I > N).
DISPLAY "Sum of 1 to " N " is " Sum.
STOP RUN.

Series-Loop.
COMPUTE Sum = Sum + I.
```

Languages are compromises...

English-like syntax	High-level (a lot happens with each instruction) Portable (same code runs the same way on completely different machines) Easy to understand Easy to code (doesn't require knowledge of the underlying machine).
Efficiency	Must trust the compiler to translate the code to efficient machine code. In practice, hand-coded assembly code, could produce an executable that runs 3 to 4 times faster.

Assembly

System software was written in assembly code, a mnemonic representation of a specific CPU's machine code. Each line representing a single machine instruction¹.

```
global _start                ; export location of _start for linker
section .text                ; code block (read-only)
_start:                      ; program starts here
    mov edx, len              ; msg length in CPU register EDX
    mov ecx, msg              ; msg address in CPU register ECX
    mov ebx, 1                ; use stdout (console output)
    mov eax, 4                ; code for SYS_WRITE (OS function)
    int 0x80                  ; signal kernel with CPU interrupt
    mov eax, 1                ; code for SYS_EXIT
    mov ebx, 0                ; return 0 to shell
    int 0x80                  ; signal kernel with CPU interrupt
section .data                ; data block (writeable)
msg db 'Hello, world!', 0xA  ; string literal
len equ $ - msg              ; len of string calculation
```

Assembly languages' compromise...

Mnemonic syntax	Low-level (each command performs a single machine instruction) Non-portable (each CPU has its own machine code) Difficult to read understand. Single thoughts are expressed over many lines of code. Hard to code (and monotonous). The example above could have been done in a single line of Python code. Requires intimate knowledge of the CPU, memory, devices, operating system.
Efficiency	Tied to the programmer's expertise. Expert programmers can "tune" the code for specific machines to achieve 100% efficient code. Can code "tricks" that the compilers don't know about.

Memory issues

Since the 1960s, programmers have had three essential choices for memory allocation:

- Static allocation (always present and fixed at compile time)
- Stack allocation (fast, but temporary)
- Heap allocation (slower to allocate/deallocate, but dynamically assigned)

Programmers faced the decision of using a high-level language like COBOL where memory allocations are completely under the control of the compiler, or assembler where there isn't a heap manager leaving us to manage all memory ourselves.

[1967 – 1972] ...then came UNIX

Originally implemented in *assembly language* for PDP-7, the UNIX operating system was created for internal use by Ken Thompson and Denis Ritchie of Bell Labs. Inspired by their work on the Multics OS for the GE-645 mainframe (a collaboration between MIT, Bell Labs, and General Electric), they wanted to reproduce as much of its ground-breaking

¹ This is not strictly true in the case of *macro assemblers*, but the focus is on coding at the level of the hardware.

ideas that they could on their smaller PDP-7 mini-computer. By 1970, much of the work was done and their new OS acquired the name, UNIX.

Unlike the majority of operating systems that were designed for business, or production use, UNIX was designed for programmers developing software and systems. What Thompson wanted next was a programming language to develop utilities for UNIX, something more economical than assembly, yet well integrated into UNIX and still efficient. After trying a Fortran compiler and abandoning it, he created a stripped-down version of BCPL, a systems programming language designed for writing compilers.

NOTATION

It was from BCPL that C inherited the curly braces notation to delimit code blocks. However, at the time few keyboards had keys for { and } so the character sequence of \$(for { and)\$ for } were used until keyboards with { } became popular.

The // comments of C++ and later C99 also come from BCPL.

Thompson simplified BCPL's syntax producing a recursive, low-level but typeless language.

NOTATION

The B language introduced us to the ++ and -- operators.

Ultimately, few programs were written in B as it was too slow and didn't support the byte addressing² of the PDP-11 CPU. Additional limitations include:

- No floating-point. BCPL had it, but the 16-bit PDP-11 did not.
- Compiles using a *threaded code*³ technique which produces slow code.

When the decision was made to port UNIX to a PDP-11 a high-level language implementation of UNIX was considered.

NB

The "new B" language existed for such a short period of time that Dennis Ritchie never wrote a full description of it. Data types were added (`int` and `char`), along with arrays and pointers.

NOTATION

Some NB declarations...

<code>int i, j;</code>	Just like C
<code>char c, d;</code>	
<code>int iarray[10];</code>	Allocates 10 integers.
<code>int ipointer[];</code>	Arrays with no dimension are pointers.
<code>char carray[10];</code>	
<code>char cpointer[];</code>	
<code>iarray[i]</code>	These are equivalent – important for C!
<code>ipointer + i</code>	

² *Byte addressability* refers to a CPU's ability to index an individual byte. The PDP-7 CPU can only address *words* (two-byte memory locations).

³ *Threaded code* is a programming technique where higher-level code is entirely converted to a series of subroutine calls. *Threaded code* has good density providing a benefit on small memory systems but may execute slower on cached systems.

[1972] What comes after B? ...C!

For UNIX to be written in a high-level language some more capabilities, thus far restricted to assembly languages, had to become available in a high-level language. Those features include:

- A richer set of mathematical operators (bitwise logic operators, shift operators, arithmetic assignment)
- Floating-point support.
- Compound data types (**objects**, but without the methods)
- A preprocessor (macros to modify the code before compiling) which helps with:
- Portability, particularly in I/O.

Performance also had to be improved. It is generally accepted that programming languages need to be space and speed efficient, produce safe and secure code, and be easy to code. It is also conventional wisdom that the more you attempt to achieve one of these goals, the more difficult it is to achieve the other two. C tries to maximize its performance capabilities while minimizing the impact on safety and ease of coding. However, when the three goals come into conflict, C favours performance over safety or ease.

How does C achieve this?

Its design philosophy: *only* support in the core language concepts that are universal and directly mappable to machine functions. As a result, C broke with most language designs and does not implement strings and I/O in the core language but pushed them into a standard library. Richie recognized that while everyone agrees strings are a fundamental necessity, they don't all agree on how they should be implemented.

As a consequence of this philosophy, C didn't acquire concurrent thread awareness in the core language until C11 in 2011. Only then, the issue of *how-best-to-do-threads* was resolved when the CPUs had support embedded in their machine code.

[1973 – 1978] C in the 1970s

By 1973, the core of C was complete as were some libraries that would become a de facto standard. UNIX's assembly code was now rewritten in C and compiled for a PDP-11. It was a success, and for the first time, an operating system was implemented using a high-level language. This feat impressed many, but being a product of AT&T Bell Labs, it couldn't be sold.

AT&T had a legal monopoly on services in United States and was barred from competing in other industries. They were a communications company, not a software company. AT&T was allowed to develop solutions internally, but they could not sell them. AT&T then distributed the UNIX source code and C compiler to universities and colleges worldwide⁴.

The popularity of UNIX and C spread and increased. C began to displace assembly for embedded and system programming tasks. The publication of *The C Programming Language* in 1978 accelerated the adoption of the language, particularly to new platforms. The book became a de facto standard for how the language and its libraries should behave.

[1979] – “C with Classes”

Danish computer scientist Bjarne Stroustrup joins AT&T Bell Labs and begins work as the head of the *Large-scale Programming Research* department. Having studied under the co-inventor of object-oriented programming (Kristen Nygaard), Stroustrup had an intimate knowledge of his object-oriented language Simula. Stroustrup knew that Simula had features useful for large software development but was too slow for practical use. Stroustrup was given the task of analyzing the UNIX kernel for distributed computing. C was good for the efficiency and integration with UNIX, but not for large-scale development. Simula was the opposite. Stroustrup set out to create a new language – based on C that

⁴ This did not include countries such as the Soviet Union which was under a technology export ban.

would add object-oriented programming from Simula, and many of the best features from ALGOL 68, Ada, CLU, and ML.

The new language called “C with Classes” would be implemented as a front-end processor (Cpre) converting the new code into standard C code. Since the front-end processor was written in C, C++ was immediately portable to any platform that supported C. If the pre-processor produced good C code, then the C compiler on the back end would produce good machine code.

“C with Classes” supported non-polymorphic classes (encapsulation), derived classes (inheritance), strong typing (unlike C), inlining, and default arguments.

[1982] – “C++”

Stroustrup starts working on a stand-alone C++ compiler (Cfront) that could translate straight from C++ code to machine code. Cfront completed the core object-oriented features of C++ by adding virtual functions (polymorphism), function and operator overloading, reference (safer and easier to use than pointers), constants, type-safe dynamic memory, better type checking, and single line comments (`//`).

1984 brought the first stream input/output library (`<iostream.h>`) that used operators rather than named functions or methods.

Following in the style of Kernighan and Ritchie, Stroustrup publishes *The C++ Programming Language* in 1985 – and yes, it starts with “Hello, world!”.

C++ 2.0 is release in 1989 along with the 2nd edition of *The C++ Programming Language* in 1991. Additions include the terrifying *multiple inheritance*, abstract classes, static member functions, const member functions, and protected members. Over the following decade templates, exceptions, namespaces, specific casts, and a Boolean primitive type.

[1983] C in the 1980s

In 1983 the American National Standards Institute (ANSI) establishes committee X3J11 to enhance the portability of C code across the growing number of platforms supporting C. C becomes the normal ‘first high-level language’ ported to a new platform. The popularity of the new personal computers from IBM and Apple with there limited memory made C the go-to language for application development.

In 1989, ANSI C – or C89 – was ratified, further helping its popularity spread. The following year, the International Standards Organization (ISO) accepts ANSI-C as an international standard. UNIX, Windows, OS/2, macOS are all now being written in C.

[1993] – template libraries

Late in 1993, Alexander Stepanov of Hewlett-Packard Labs presents a templated library to the C++ standardization committee. Stepanov had demonstrated how templated classes and functions could be used to created *generic abstract data types* and *algorithm* without compromising efficiency. This work would become the *Standard Template Library*.

[1995 – 1999] C in the 1990s

C95 brings support for Unicode in 1995. But complaints about the inconsistency of extensions to C needed to be addressed. Through the 90s, the ISO worked out bug fixes, and added new features, culminating in a major update to C in 1999. For the first time C borrowed new features from C++. Surprisingly, Microsoft breaks from the pack and doesn’t support C99, instead putting all their support behind C++. Microsoft’s Visual C++ can still compile C code, but with the older C90 rules.

C++ Features in C99

- The inline keyword
- Declarations can now be mixed with code (not all at the top).
- For-loops can now declare the loop variable in the initialization clause.
- Double-slash // comments
- Universal character names in the source code.

[1998] Standard C++

C++ becomes an ISO standard incorporating the additions of the 90s and Stepanov's *template library*. The *template-approach* radically transforms the standard library as it is utilized throughout the library. The majority of the C standard library was also ingested but placed inside the namespace *std*.

In contrast to C, the standard library is growing acquiring many new libraries.

[2000s] C & C++ in the 2000s

The C and C++ committees try to pull the two languages back together where they can and work towards making all shared features behave the same way.

C11 (2011) becomes thread-aware and includes some controversial 'safe' function alternatives. C17 – the latest ratified standard adds nothing new but addresses bugs since C11.

C++ 11 (2011) becomes thread-aware and accepts many new libraries from the BOOST organization⁵, an open-source community for C++ libraries.

While the C committee takes a break from new features, the C++ committee commits to new developments every 3 years, releasing updates to the standard in 2014 (C++14), 2017 (C++17), and 2020 (C++20). Their approach is one of constant development, but on no specific due date. Features are discussed, refined, and whatever is ready by the next standard is incorporated.

[2018] Draper prize

The Draper prize is awarded by the U.S. National Academy Engineering, to Bjarne Stroustrup for the creation of C++. It is one of three prizes that constitute the "Nobel Prizes of Engineering", the others are the Academy's Russ and Gordon Prizes.

[2020s] Future C/C++

C++ 20 was a big change. Some have worried that they did too much in the last three years, but some of the features had been under development for more than twenty years. C++20 was big, simply because many of their projects came to completion in time for the release. Already underway, C++20 features will take some time to be fully implemented in commercial compilers.

New C++20 features

- Concepts
- Modules
- Designated initializers (from C)
- More advanced lambdas
- The spaceship operator <=>
- New attributes
- More constexpr
- constexpr
- a revised memory model (resolves issues with Power, ARM, and GPU implementations)
- coroutines

⁵ <https://www.boost.org/>

- constinit
- ranges
- atomic smart pointers
- calendars for <chrono>
- bit_cast<>

Working draft – C23

C23 currently in draft is expected to include:

- Decimal floating-point types
- More C++ compatibility
- C++ attributes [[nodiscard]], [[maybe_unused]], [[deprecated]], and [[fallthrough]]
- New literal constants, including C++'s binary integer (`int x = 0b101010;`)
- The usual bug fixes...

Working draft C++23 (or later)

In the works are the following:

- Contracts
- Reflection
- Metaclasses
- Executors
- Networking extensions
- Properties
- Extended futures (concurrency)

Description, Philosophy, and Influence

C

Description

C is a general-purpose, imperative (procedural) structured, computer programming language. It emphasizes system, embedded, utility, library, and high-performance programming.

- Lexical variable scope.
- Recursion.
- Static type system.
- Low-level access to memory.
- Inline assembly code.

Philosophy

The principle of design that C seems to espouse is that the core language should not implement any feature that is not directly mappable to a machine instruction found on the majority of CPUs. Anything that is not universal should be implemented in a library. Minimal runtime support should be a priority.

This keeps the core language small, easy to port, and easier to optimize.

Influence

C's greatest impact is on syntax; how languages group blocks of code (curly braces), declare variables (`int i;`), common operators (`%` for mod, `++` for increment).

Languages that in some way derive from C include:

- Ch
- CINT
- C++
- Objective-C
- Java
- JavaScript
- C#
- D
- Go
- Python
- PHP
- Perl
- Limbo
- LPC
- C shell
- Cilk

And hundreds more...

C++

Description

C++ is a general-purpose, multi-paradigm programming language, supporting imperative (procedural), functional, object-oriented, generic, and modular paradigms. Like C, it emphasizes system, embedded, utility, library, and high-performance programming, but provides additional strength in the areas of infrastructure, resource-constrained, and large-scale programming.

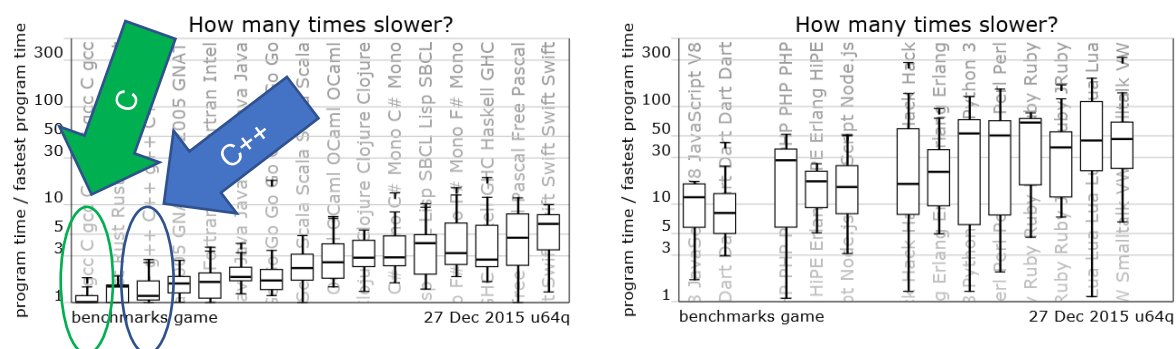
Philosophy

The design philosophy or “rules of thumb” are outlined by Stroustrup in *Evolving a language in and for the real world* (2007). They are:

The rule	Commentary
General rules:	
C++ evolution must be driven by real problems.	Let the world drive the language; not the language drive the world.
Don't get involved in a sterile quest for perfection.	
C++ must be useful now.	
Every feature must have a reasonably obvious implementation.	Be practical, don't pursue 'theoretical' possibilities.
Always provide a transition path.	Don't make programmers rewrite everything to use a new feature.
C++ is a language, not a complete system.	Let the programmer express their idea, not the language's idea.
Provide comprehensive support for each supported style.	Don't dictate how a programmer should solve their problem.
Don't try to force people to use a specific programming style.	
Design support rules:	
Support sound design notions.	
Provide facilities for program organization.	Help programmers manage complexity.
Say what you mean.	Don't hide side-effects, consequences.
All features must be affordable.	Don't do what you are not good at.
It is more important to allow a useful feature than to prevent every misuse.	Don't block good programmers, because of poor programmers' failings.
Support composition of software from separately developed parts.	Basic economics: multiple small teams is more cost effective than huge teams, components can reused, etc.
Language-technical rules:	
No implicit violations of the static type system.	Don't place arbitrary restrictions on what the programmer can store and where, but also don't surprise them.
Provide as good support for user-defined types as for built-in types.	Allow programmers to build the language without restriction.

The results change over time: I've provided the results for 2015 below. Note that the general trend is the same. But a couple of trends:

1. Newer languages make the most improvement: as they mature, their optimizations improve. However, further optimizations become more difficult. Don't expect the trends to continue.
2. The range of results narrow (i.e., the bars are shorter). This also indicates that the optimizations are maturing.



C++ now beats C: most benchmarks still place C at the front. And there is an error in the C++ implementation (the delete was not an array delete). I'm not certain how much this would impact the results, if at all. Nonetheless, it is a result in C++'s favour.

A grain of salt: you must be cautious when reading benchmarks. They can be influenced by many factors:

- Which compiler is used (some perform better on different operating systems)?
- What operating system is used? (e.g.: Windows generally provides more runtime checks than Linux does)
- What is running in the background? (this can affect the availability of cores, time to switch tasks)
- Hardware differences (CPU vs. memory speed)
- Type of benchmark (is the benchmark memory intensive, compute intensive, integer or floating-point, or a mix)

So many factors affect the outcome of a benchmark. You need to look at multiple benchmarks, on multiple systems, over generations of the language. Currently, C and C++ have demonstrated the highest performance levels over the longest period.

Some other benchmarks:

Programming Language Performance Comparison (hildstrom.com)

Programming Languages Benchmarks (attractivechaos.github.io)

Popularity

How do you measure popularity?

- Amount of code written. Or being written now?
- Number of systems develop with it. Or being developed with it?
- Most liked?
- Most taught?
- Least complaints?
- Popular opinion?
- Expert opinion?

- Most jobs?
- Most critical (can it be easily replaced)?

Popularity statistics often say more about the person doing the measurement, than what is being measured.

The Tiobe Index

The Tiobe index (<https://www.tiobe.com/tiobe-index/>) tracks popularity by aggregating statistics on the number of skilled engineers world-wide, courses and third-party vendors, as well as activity on search engines. It doesn't measure the quality or importance of a language but tries to measure usage.

Here are Tiobe's top five for 2023 and 2020:

Language	Dec 2023		Dec 2020	
	Rank	Rating	Rank	Rating
Python	1	13.86%	3	12.21%
C	2	11.44%	1	16.48%
C++	3	10.01%	4	6.91%
Java	4	7.99%	2	12.53%
C#	5	7.3	5	4.20%

*Note that all of the top 5 are C-derived languages.



















The big take-away... the top languages don't change much from year to year, and C and C++ have been in the top five every year since the site was created.





IEEE Index

The Institute of Electrical and Electronic Engineering also maintains an [index](#). Their metrics put more weight on surveys of their membership, and separate languages into development domains. Afterall, a great language for embedded development may be irrelevant to web development, and vice versa. You can grossly interpret their results as the answer to the question, "what languages should a developer know?"

Since the site is paywalled, I'll focus on last year's results (which are little different from this year).

Overall







Rank	Language	Type	Score
1	Python	  	100.0
2	Java	  	96.3
3	C	  	94.4
4	C++	  	87.5
5	R		81.5
6	JavaScript		79.4
7	C#	   	74.5




 =web,  =mobile,  =Enterprise,  =embedded

In 2020, C# moved up into 5th place.






The index is more interesting when restricted to a language type:

Mobile






Rank	Language	Type	Score
1	Java	  	96.3
2	C	  	94.4

3	C++		87.5
4	C#		74.5
5	Swift		69.1

Enterprise

Rank	Language	Type	Score
1	Python		100.0
2	Java		96.3
3	C		94.4
4	C++		87.5
5	R		81.5

Embedded

Rank	Language	Type	Score
1	Python		100.0
2	C		94.4
3	C++		87.5
4	C#		74.5
5	Arduino		67.2

What about web?

C and C++ are not considered web languages, but that doesn't mean they are not used in website development. Websites are often front-ends for large enterprise system, or electronic devices, both of which are often developed in C or C++.

Employment

Just how many people are actively developing in C/C++? It can be difficult to measure:

- Who decides if you program enough to be considered a 'developer?' (are educators, students, and hobbyists, developers?)
- How do you enumerate developers that work in multiple languages?
- How much time do you have to spend programming a language to be considered 'active?'
- Do you have to develop commercial software with a language to count as an 'active developer' in the community?

Very difficult, to say the least. But the people at [/DATA](#) (slash-data) have perhaps done the best work in trying to answer these questions.

In 2021, they reported that the world programming contains approximately, 24.3 million active developers. Of that number:

- 13.8 million worked in JavaScript
- 10.1 million worked in Python
- 9.4 million worked in Java
- 7.3 million worked in C/C++
- 6.5 million in C#
- 6.3 million in PHP

The two closest systems languages were:

- 1.6 million using Objective-C
- 1.3 million using Rust

They did indicate that the Rust numbers don't reflect the amount of commercial work being done.

So, some perspective... C/C++ account for ¼ of the world's developers and is a much larger community than all the other systems language communities *combined!*

Notable Uses of C & C++

Here is a sample of the types of software developed with C and C++.

- Applications with systems components
- Banking and financial (funds transfer, financial modeling, customer interaction, teller machines, ...)
- Embedded Systems (instruments, cameras, cell phones, vehicles, appliances, medical devices, ...)
- Games
- GUI
- Graphics
- Hardware design and verification
- Low-level system components (device drivers, network layers, security software, ...)
- Scientific and numeric computation (physic, engineering, simulations, data analytics, ...)
- Servers (web servers, large application backbones, billing systems, ...)
- Small business applications (inventory, customer service, ...)
- Symbolic manipulation (geometric modeling, vision, speech recognition, ...)
- Systems programming (compilers, operating systems, database systems, ...)
- Telecommunications (phones, networking, monitoring, billing, operating systems, ...)

And some specific cases:

Category	Products
Operating Systems	UNIX (C, assembly language) Windows 10 (C, C++, C#) Linux (C, assembly language) OS X (C, C++, Objective-C, Swift, assembly language) iOS (C, C++, Objective-C, Swift, assembly language) Android (C :core, Java :UI, C++ :most of the rest) Sybian (C++) IBM K42 (C++) – a very high-end operating system IBM AS/400 (C++) Java JVM (C, C++)
Embedded Systems	Carputers ⁶ (automatic transmission, tire pressure detection systems, sensors, seat controls, dashboard display, anti-lock brakes, etc.) Vending Machines Credit card readers BIOSes Lockheed-Martin Aero: airplane control of F16, JSF, ... (C++)
Programming Languages	Python (C) Ruby (C) Perl (C) Lua (ANSI C)

⁶ Carputers are computers that are embedded into vehicles.

	Tcl (C) – often embedded into C and C++ programs. AWK (C) BASH (C) Julia (Julia, C, C++, Scheme)
Programming Libraries	OpenGL MATLAB GNU Scientific Library libuv (Node.js)
System Software	Chromium (C, C++, Python, Java, JavaScript) Firefox (C, Rust, Javascript, HTML, C++) Device drivers – almost everyone (C, C++)
Enterprise Systems	Amadeus – airline reservations (C++) Amazon – e-commerce (C++) Vodafone – mobile phone infrastructure (C++)
Graphic Applications	VLC Media Player (C, C++, Objective-C) Maya – 3D animation (C++) 3DStudioMax (C++)
Games {Engines }	Assassin’s Creed {Anvil: C++, C#} Battlefield {Frostbite: C++} Call of Duty franchise {IW engine: C++} Civilization IV {Gamebryo: C++} Crysis {CryEngine: C++} Devil May Cry 4 {MT Framework: C++} Doom {id Tech 1-3: C} Doom 3 {id Tech 4-7: C++} Europa Universalis {Clausewitz: C++} Fallout {Creation Engine: C++} Far Cry {CryEngine: C++} FIFA {Frostbite: C++} Final Fantasy X {PhyreEngine: C++} Grand Theft Auto {RenderWare: C++} Half-Life {GoldSrc: C, C++, assembly} Half-Life 2 {Source: C++} Halo franchise (C++) Lego Universe {Gamebryo: C++} Madden NFL {Frostbite: C++} Mass Effect {Frostbite: C++} Morrowind {Gamebryo: C++} Oblivion {Gamebryo: C++} Pokémon Go {Unity: C++} Prince of Persia{ Anvil: C++, C#} Skyrim{Creation Engine: C++} Star Wars: The Old Republic {HeroEngine: C++, C#} Unreal {Ureal: C++} World of Warcraft (C++)
Development Tools	Git (C, Shell, Perl, Tcl, Python) Intel chip design and manufacturing (C++) Microsoft Visual Studio (C++) Java (C++) GCC (C, C++) Node.js (C, C++, JavaScript) TensorFlow (Python, C++, CUDA)
Application Software	Microsoft Office (Originally C, now C++) Adobe Suite (mostly C, C++)

	Mathematica (C++)
Database Software	MySQL (C, C++) Oracle (C, C++, assembly language) MS SQL Server (C, C++) PostgreSQL (C) Google: search engine, Google Earth (C++)

Criticisms of C and C++

“There are only two kinds of languages: the ones people complain about and the ones nobody uses.” – Bjarne Stroustrup

Here are the some of the most common criticisms of C and C++, with commentary.

C

Using integers as Booleans: makes the precedence of operations on ‘bools’ unintuitive.

- Fixed in C++

No Call-By-Refence: reference parameters are handled by pointer, so the compiler can’t assume that the variable is at the received memory location, whereas proper call-by-refence will have the compiler check at the call site.

- Fixed in C++

Too much undefined behavior: code can be written that has behavior that is undefined, like when a function has a control path that doesn’t return a value.

Missing features: where are the graphics, networking, etc.?

- C consciously does not inflict platform specific solutions that are not universal. Graphics has more to do with the platform than the language.
- There are thousands of libraries that support what is missing.
- Much of what people ask for (like garbage collecting) would make C into Java or C#.

Automatic memory management: memory leaks, memory overruns, memory ...

- This is a problem, but not a clear solution as the point of C is control and efficiency. Automating memory management adds a layer of complexity (and processing) to the code. There are some problems like unions that don’t easily or efficiently lend itself to memory management.
- Or you could just use Rust and take the performance hit (although small).

C++

The language is too big: most project teams don’t use all the features of C++.

- Nobody uses all the features of any language, why is this a problem just for C++?
- The multi-paradigm approach allows a programmer to stay within the language. The alternative would be to use a multi-language approach, or to restrict yourself to one paradigm and hope you can figure out how to implement the solution. C++ promotes the idea that you write the code you want, not what the compiler wants.

It’s too complex: the features are too complex for a programmer to use properly.

- Does this say more about the critic or the language?
- It wasn’t designed to be everything to everybody, but as a programming language that can model difficult designs.

Slow compile times: agreed! Changes to header file can trigger long recompiles.

- Modules (C++ 20) alleviates much of this problem. In C++ 20, the module import command “import std;” is the equivalent of #including every standard library header, yet it compiles faster than #including <iostream> alone.

Global format state of <iostream>: yes. Exceptions can prevent a restoration of object state.

- You can attach a scoped referencing object that will restore the original state on scope release, but it is more work for the programmer.

The two-iterators problem: using two separate variables (iterators) to represent a single range is odd and error prone.

- Ranges (C++ 20) fix this issue.

Exception overhead: compiling with exceptions produces larger binaries due to the extra code to unravel the stack on clean up and process all the exit points.

- You can turn it off.
- You can mark functions as ‘nothrow’.
- You can recognize that it is doing the clean up you would have to do in any case – C++ exceptions are no worse than any other language implemented exceptions.

Encoding literals in source code: C++ doesn’t validate the character encoding with regard to the output file or device.

- This one is on the programmer to resolve.

Analysis of criticism

Why hasn’t the criticism caused a decline in use? Before you switch language, you must answer the following questions:

- Can the new language fully replace C or C++? Is there any loss in capability? For example: moving from C++ to Java would remove the functional programming paradigm from your solution, it would require a significant run-time to be installed, and it can directly access hardware.
- C and C++ are multi-vendor, standardized languages? Is the new language proprietary, can you trust the one company to support you in the future?
- Will your staff be able to code in the new language? Replacing veteran staff with new staff usually results in a loss of knowledge about the platform, application, client, environment, etc.
- What is the cost of switching? Will the expense be justified by either *reduced costs*, or *increased revenues*?
- What problems will the new language introduce? C and C++ do have problems, but they are well understood, and solutions have been developed for them. New languages may not have their problems identified, or not have a work around yet developed.
- All languages change. Will the feature you don’t like, be fixed in an upcoming release of your existing language?
- Does the new language have a proven track record that you can count on? Has it proven itself? Is it here for the long haul? Just ask developers of PowerBuilder or ColdFusion. Any veteran programmer can provide a list of the languages they no longer use.

When to Use C?

Consider using C for:

- General purpose programming of small to mid-size applications.
- High-performance software.
- Software requiring direct access to hardware.
- Systems with limited resources.

- When the only other alternative is assembly language.

When to Use C++?

“Python where you can, C++ where you have to...” – programming mantra, University of Waterloo.

Consider using C++ for:

- Anything you would use C for...
- Large-scale systems
- Where architecture is a dominant element of the solution.

Why Learn C?

- Lingua Franca of programming
 - Most new ideas in program are expressed first as C code.
 - Most new libraries arrive in C first (and sometimes only).
 - Bridge language: **Python**, Fortran, etc. can directly call C functions.
 - Every platform
- Understanding C is understanding the machine.
- Top 5 language for each of the last 50 years.
- It's a stepping stone to learning C++.
- Ubiquitous language:
 - infrastructure is C.
 - Nothing but C++ can replace it (because C++ is a better C)

Why Learn C++?

All the reasons you would learn C, and:

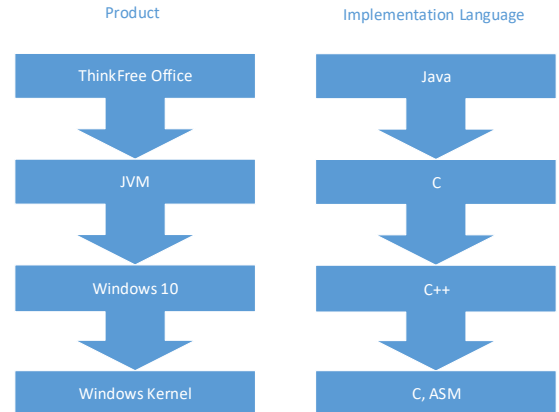
- Top 5 language for each of the last 40 years.
- Can gradually move into other paradigms of programming.
- Can easily mix multiple paradigms without having to switch to a different language.

The Future of C and C++

They aren't going anywhere...

- C was updated in June 2018 – C18.
- C++ was updated in December 2020 – C++20.
- C23 is finished and currently working its way into compilers.
- C++23 is currently being developed.

It is almost impossible to run a program without some C or C++ code running along with it. The diagram to the right shows the layers of software that run when a Java application executes on a Windows computer. Three of the four layers are either C or C++. And those layers are the most difficult to replace.



The explosion in Internet-of-Things devices, the competition for faster video games, mobile applications, cryptocurrency, power efficiency, machine-learning, and artificial intelligence – they all rely on C and C++ to get the work done that the other languages can't. Will Rust, Go, or Swift, or some future language one day replace them. In the short term, the answer is a definitive no. In the longer term, it's an "I don't know". We see a little bit of movement toward Rust from C but not from C++. None have yet demonstrated that they can do the heavy lifting C++ does.

So what will replace C++20? C++23! What will replace C17? C23. Beyond that we just don't know.

Garth Santor & Trinh Hân
December, 2023

Document History

1.0.1	2020-12-26	Created
1.1.0	2021-05-04	Add section on total number of active programmers.
1.2.0	2023-12-28	Updated some statistics,