Doc. No. P1947

Doc. No. P2317R0
Date: 2021-02-14
Audience: All

Reply to: Bjarne Stroustrup (bs@ms.com)

C++ – an Invisible Foundation of Everything

I wrote this short note to summarize some principles of and ideals for C++ them based on documents outlining the design and evolution of C++ with links to those documents. As published in ACCU Overload No.161, it is just 4 pages.

Several people asked me "what's the intended audience for this paper?" and "did you write it for the WG21 members?" I didn't really have a specific audience in mind. Some WG21 members might find it useful so I add it to out "mailings." My hope was and is that this paper will have a long "shelf life" so that it will turn up in contexts that I can't imagine and be a positive contribution to the interminable and often totally confused discussions about what C++ is and should be.

C++ – an Invisible Foundation of Everything

Bjarne Stroustrup
Morgan Stanley, Columbia University
www.stroustrup.com

I am often asked variations of the questions "What is C++?" and "Is C++ still used anywhere?" My answers tend to be detailed, focused on the long term, and slightly philosophical, rather than simple, fashionable, and concrete. This note attempts a brief answer. It presents C++ as "a stable and evolving tool for building complex systems that require efficient use of hardware." Brief answers are necessarily lacking in depth, subtlety, and detail – for detailed and reasoned explanations backed by concrete examples, see the references and resources (§7). This note is mostly direct or paraphrased quotes from those sources.

1. Overview

This note consists of

- *§2. Aims and means* – the high-level aims of C++'s design and its role in systems
- *§3. Use* – a few examples of uses of C++ focusing on its foundational uses
- *§4. Evolution* – the evolutionary strategy for developing C++ based on feedback
- *§5. Guarantees, Language, and Guidelines* – the strategy for simultaneously achieving evolution, stability, expressiveness, and complete type-and-resource safety
- *§6. People* – a reminder of the role of people in software development
- *§7. References and resources* – an annotated list of references that can lead to a deeper understanding of C++
- *Appendix* – a very brief summary of C++'s key properties and features

2. Aims and means

C++ was designed to solve a problem. That problem required management of significant complexity and direct manipulation of hardware. My initial ideals for C++ included

- **The efficiency of C for low-level tasks**
- **Simula's strict and extensible type system**

What I did not like included

- **C's lack of enforcement of its type system**
- **Simula's non-uniform treatment of built-in types and user-defined types** (classes)
- **Simula's relatively poor performance**
- **Both languages' lack of parameterized types** (what later became templates)

This set off a decades-long quest to simultaneously achieve

- **Expressive code**
- **Complete type-and-resource safety**
- **Optimal performance**

I did not want a specialized tool just for my specific problem (support for building a distributed system), but a generalization to solve a large class of problems:

- **C++ is a tool for building complex systems that require efficient use of hardware**

That's a distillation of my initial – and current – aims for C++. Suitably fleshed out with details and implications, this explains much about modern C++. That statement is not a snappy slogan of the form ***C++ is an <<adjective>> language*** but I have never found a sufficiently accurate and descriptive adjective for that. Shifting the focus from language use to language technicalities, we can say:

- **C++ is a general-purpose language for the definition and use of light-weight abstractions**

That leaves the definition of "general-purpose", "light-weight", and "abstraction" open to debate. In C++ terms, I am primarily thinking about classes, templates, and concepts; about expressiveness and efficient use of time and space.

To elaborate a bit further:

- **C++ supports building resource-constrained applications and software infrastructure**
- **C++ supports large-scale software development**
- **C++ supports completely type-and-resource-safe code**

Technically, C++ rests on two pillars:

- **A direct map to hardware**
- **Zero-overhead abstraction in production code**

By "zero-overhead" I mean that roughly equivalent functionality of a language feature or library component cannot by expressed with less overhead in C or C++:

- **What you don't use, you don't pay for** (aka "no distributed fat")
- **What you do use, you couldn't hand-code any better** (e.g., dynamic dispatch)

It does not mean that for a more-specific need you can't write more efficient code (say in assembler).

3. Use

Many well-known applications/systems are written in C++ (e.g., Google search, most browsers, Word, the Mars Rovers, Maya). All systems need to use hardware and large systems must manage complexity. Supporting those fundamental needs has allowed C++ to prosper over decades :

- **C++ is an invisible foundation of everything**

"Everything" is obviously a bit of an exaggeration, but even systems without a line of C++ tend to depend on systems written in C++. "Everything" is a good first approximation.

Foundational uses of C++ are typically invisible, often even to programmers of systems relying on C++: to be usable by many, a complex system must protect its users from most complexities. For example, when I send a message, I don't want to know about message protocols, transmission systems, signal processing, task scheduling, processor design, or provisioning. Thus, we find C++ in virtual machines (HotSpot, V8), numerics (Eigen, ROOT), AI/ML (TensorFlow, PyTorch), graphics and animation (Adobe, SideFx), communications (Ericsson, Huawei, Nokia), database systems (Mongo, MySQL), finance (Morgan Stanley, Bloomberg), game engines (Unity, Unreal), vehicles (Tesla, BMW, Aurora), CAD/CAM (Dassault, Autodesk), aerospace (Space-X, Lockheed Martin), microelectronics (ARM, Intel, Nvidia), transport (CSX, Maersk), biology and medicine (protein folding, DNA sequencing, tomography, medical monitoring), embedded systems (too many to mention), and much more that we never see and typically don't think of – often in the form of libraries and toolkits usable from many languages. C++ is also key in components and implementations of many different programming languages (GCC, LLVM).

We also find C++ in "everyday" applications, such as coffee machines and pig-farm management. However, the role as a foundation for systems, tools, and libraries has critical implications for C++'s design, use, and further evolution.

4. Evolution

Since its inception, C++ has been evolving. That reflects both necessity and an early deliberate choice:

- **No language is perfect for everything and for everybody** (that includes C++)
- **The world changes** (e.g., there were no mobile apps until about 2005)
- **We change** (e.g., few industrial programmers appreciated generic programming in 1985)

Thus

- **C++ must evolve to meet changing requirements and uses**
- **Design decisions must be guided by real-world use** – all good engineering relies on feedback

To evolve, C++ must

- **Offer stability** – organizations that deliver and maintain systems lasting for decades can't constantly rewrite their systems to keep up with incompatible changes to their foundations.
- **Be viable at all times** – must be effective for problems in its domain at all times; you can't take a "gap year" from improving the language and its implementation.
- **Be directed by a set of ideals** – to remain coherent, the development of language features must be guided by a framework of principles and long-term aims.

Why continue to evolve after years of success? There never was a shortage of people who would prefer to stay with C or move to one of the latest fashionable languages. People can to do exactly that if it makes sense to them, but

- **C++ is a good solution to a wide range of problems**
- **There are hundreds of billions of lines of working C++ code "out there"**
- **There are millions of C++ programmers**

It takes significant time for a language to mature to be adequate for a range of uses far beyond the understanding of its original designers. Some design tensions are inherent

- **Every successful language will eventually face the problem of evolution vs. stability**
- **Every general-purpose language must serve both (relative) novices and seasoned experts**

Successful language design – like all successful engineering – requires good fundamental ideas and a careful balancing of constraints. Optimizing for just a single desirable property can offer advantages for one application area for one moment of time, but eventually the result dies for lack of adaptability. By now, C++ has survived for 40 years by carefully balancing concerns, learning from experience, and avoiding chasing fashions.

- **A general-purpose language must maintain a careful balance of user needs**

Essential concerns that must be balanced include:

- **simplicity, expressiveness, safety, run-time performance, support for tool building, ease of teaching, maintainability, composability of software from different sources, compilation speed, predictability of response, portability, portability of performance, and stability**

"Simplicity" refers to how ideas are expressed in source code, "expressiveness" determines the range of uses, "safety" to type safety and absence of resource leaks, and "predictability" is essential for many embedded systems.

5. Guarantees, Language, and Guidelines

C++ is complicated, but people don't just want a simpler language, they also want improvements and stability:

- **Simplify C++**
- **Add these new features**
- **Don't break my code**

These are reasonable requests so we need a way out of this "trilemma." We cannot simplify the language without breaking billions of lines of code and seriously disrupt millions of users. However, we can dramatically simplify the use of C++:

- **Keep simple tasks simple**
- **Ensure that nothing essential is impossible or unreasonably expensive**

To do that

- **Provide simpler alternatives for simple uses**
- **Provide simplifying generalizations**
- **Provide alternatives to error-prone or slow features**

Often, a significant improvement involves a combination of those three.

- **Design C++ code to be tunable**

A high-level abstraction presents a simple, safe, and general interface to users. When needed, a user – not just a language implementer – can provide an alternative implementation or an improved solution. This can sometimes lead to orders-of-magnitude performance improvements and/or enhanced functionality. By using lower-level or alternative abstractions, we can eventually get to use the hardware directly, sometimes even to directly access special-purpose hardware (e.g., GPUs or FPGAs).

From the earliest days, one major aim for the evolution of C++ was to deliver

- **Complete type-and-resource safety**

Much of the evolution of C++ can be seen as gradually approaching that ideal, starting with adding function declarations (function prototypes) to C. By "type safety" I mean complete static (compile-time) checks that an object is used only according to its defined type augmented by guaranteed run-time checks where static checking is infeasible (e.g., range checking). Simula offered that but at significant cost implying lack of applicability in key areas.

- **Making the type system both strict and flexible is key to correctness, safety, and performance**

Type-safety is not everything, though:

- **Correctness, safety, and performance are system properties, not just language features**
- **A type-safe program can still contain serious logic errors**
- **Test early, often, and systematically**

To simplify use, we need tools and guidelines. The "C++ Core Guidelines" (see the references and resources; §7) offer rules for simple, safe, and performant use:

- **No resource leaks** (incl. no leaks of non-memory resources, such as locks and thread handles)
- **No memory corruption** (an essential pre-condition for any guarantee)
- **No garbage collector** (to avoid indirections in access, memory overheads, and collection delays)
- **No limitation of expressiveness** (compared to well-written modern C++)
- **No performance degradation** (compared to well-written modern C++)

These guarantees cannot be provided for arbitrarily complex C++ code. Therefore, the Core Guidelines include rules to ensure that static analysis can offer the needed guarantees. The guidelines are a key part of my strategy for a gradual evolution of C++:

- **Improve C++ by adding language features and libraries**
- **Maintain stability/compatibility**
- **Provide a variety of strong guarantees through selectively enforced guidelines**

The Core Guidelines are in production use, often supported by static analysis. The guidelines can be enforced by a compiler, but the aim is not to impose a single style of use on the whole C++ community. That would fail because of the widely varying needs and styles of use. By default, enforcement must be selective and optional. A separate static analyzer – usable with any ISO C++ compatible implementation – would be ideal. If a specific "dialect" (that is, a specific set of rules and enforcement profiles) is to be enforced, it can be done through control of the build process (possibly supported by compiler options).

6. People

Code is written by people. A programming language is a tool, just one part of a tool chain for a technical community. This was recognized from the start. Here is the opening statement of the first edition of "The C++ Programming Language":

- **C++ is a general-purpose programming language designed to make programming more enjoyable for the serious programmer**

By "serious programmer" I meant "people who build systems for the use of others." This concern for the human side of system development has also been express as:

- **Design and programming are human activities; forget that and all is lost**

C++ serves a huge community. To improve software, we need not just to improve the language. We must also bring the community along – supported by education, libraries, and tools. This must be done carefully because no individual can know every use of C++ or every user need.

7. References and resources

- B. Stroustrup: Thriving in a crowded and changing world: C++ 2006-2020. ACM/SIGPLAN History of Programming Languages conference, HOPL-IV. June 2020. This is the best current description of C++'s aims, evolution, and status. At 160 pages, it is not a quick read.
- H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong: DIRECTION FOR ISO C++ . WG21 P2000. 2020-07-15. Outlines the direction of C++'s evolution, co-authored and continuously updated by the ISO C++ Standard committee's Direction Group as a guide to members.
- B. Stroustrup: The Design and Evolution of C++. Addison Wesley, ISBN 0-201-54330-3. 1994. This book contains lists of design rules for C++, some early history, and many code examples.
- B. Stroustrup: A Tour of C++ (2nd Edition). ISBN 978-0134997834. Addison-Wesley. 2018. A brief – 210 page – tour of the C++ Programming language and its standard library for experienced programmers.
- B. Stroustrup: Programming -- Principles and Practice Using C++ (2nd Edition). Addison-Wesley. ISBN 978-0321992789. May 2014. A programming text book aimed at beginners who want eventually to become professionals.
- The C++ Core Guidelines. A set of guidelines for safe and effective use of modern C++. Many of the guidelines are enforceable through static analysis. 2014-onwards.
- Infographic: C/C++ Facts We Learned Before Going Ahead with CLion. A 2015 report on a survey of C++ use, estimating the C++ user community to be 4.5 million strong and listing major industrial use. Today, there are more users.
- B. Stroustrup, H. Sutter, and G. Dos Reis: A brief introduction to C++'s model for type- and resource-safety. Isocpp.org. October 2015. An early summary of the aims of the core guidelines as they relate to type safety and resource safety.
- B. Stroustrup: How can you be so certain? P1962R0. 2019-11-18. A caution against shallow arguments for fashionable causes. Language design requires a certain amount of humility.
- B. Stroustrup: Remember the Vasa! P0977r0. 2018-03-06. A note of warning about overenthusiastic "improvement" of the language.
- The C++ Foundation's Website describes the organization and progress of the standards effort.

- www.stroustrup.com offers many of my videos, papers, interviews, and quotes, including
  - My CppCon'14 keynote: Make Simple Tasks Simple!
  - My Cppcon'17 keynote: Learning and Teaching Modern C++
  - My Cppcon'19 Keynote: C++20: C++ at 40
  - Lex Fridman's 2019 Interview with Bjarne Stroustrup

Appendix: The C++ language

The description of C++ above does not mention any language features or give any code examples. This leaves it open to serious misinterpretation. I cannot give serious examples of good code here – see the references (§7) – but I can summarize.

There is a reasonably stable core of ideals that guides the evolution of C++ (the references are to my 2020 History of Programming Languages paper):

- **A static type system with equal support for built-in types and user-defined types (§2.1)**
- **Value and reference semantics (§4.2.3)**
- **Systematic and general resource management (RAII) (§2.2)**
- **Support for efficient object-oriented programming (§2.1)**
- **Support for flexible and efficient generic programming (§10.5.1)**
- **Support for compile-time programming (§4.2.7)**
- **Direct use of machine and operating system resources (§1)**
- **Concurrency support through libraries (often implemented using intrinsics) (§4.1) (§9.4)**

Key language features with their primary intended roles:

- **Functions** – the basic way of defining a named action. Functions with different types can have the same name. The function invoked is then chosen based on the type of its arguments.
- **Overloading** – allowing semantically similar operations on different types is a key to generic programming.
- **Operator overloading** – a function can be defined to give meaning to an operator for a given set of operand types. Overloadable operators includes the usual arithmetic and logical operators plus **()** (application), **[]** (subscripting), and **->** (member selection).
- **Classes** – user-defined types that can approach built-in types for ease of use, style of use, and efficiency, while opening up a whole new world of general and application-specific types. Classes offer (optional) encapsulation without run-time cost. Class objects can be allocated on the stack, in static memory, in dynamic (heap) memory, or as members of other classes.
- **Constructors and destructors** – the key to C++'s resource management and much of its simplicity of code. A constructor can establish an invariant for a class and a destructor can release any resources an object has acquired during its lifetime. Systematic resource management using constructors and destructors is often called RAII ("Resource Acquisition Is Initialization").
- **Class hierarchies** – the ability to define one class in terms of another so that the base class can be used as an interface to derived classes or as part of the implementation of derived classes. The key to traditional object-oriented programming.
- **Virtual functions** – provide run-time type resolution within class hierarchies.

- **Templates** – allow types, functions, and aliases to be parameterized by types and values. The workhorse of C++ generic programming.
- **Concepts** – compile-time predicates on sets of types and values. Mostly used as precise specifications of a template's requirements on its parameters, thereby allowing overloading. A concept taking a single type argument is roughly equivalent to a type, except that it does not specify object layout.
- **Function objects** – objects of classes (often class templates) supporting an application operator **()**. Acts like functions but are objects that can carry state.
- **Lambdas** – a notation for defining function objects.
- **Immutability** – immutable objects can be defined. Access through pointers or references can be declared to be non-mutating.
- **Modules** – a mechanism for encapsulating a set of types, functions, and objects with a well-defined interface offering good information hiding. To use a module, you **import** it. A program can be composed out of modules.
- **Namespaces** – for separating major components of a program and avoiding name clashes.
- **Exceptions** – for signaling errors that cannot be handled locally. The backbone of much error handling. Exceptions are integrated with constructors and destructors to enable systematic resource management.
- **Type deduction** – to simplify notation by not requiring the programmer to repeat what the compiler already knows. Essential for generic programming and simple expression of ideas.
- **Compile-time functions** – part of comprehensive support for compile-time programming.
- **Concurrency** – lock-free programming, threads, and coroutines.
- **Parallelism** – parallel algorithms.

In addition, there is a relatively large and useful standard library and loads of other libraries. Don't try to write everything yourself in the bare language.

Acknowledgements

This note is mostly direct or paraphrased quotes from the referenced papers, so many thanks to the contributors to those as listed in their acknowledgement sections. Also thanks to Gabriel Dos Reis, J.C. van Winkel, Herb Sutter, J-Daniel Garcia, Roger Orr and the unnamed Overload reviewers who made constructive comments on earlier drafts.