C++

C++ (/ˌsiːˌplʌsˈplʌs/) is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or "C with Classes". The language has expanded significantly over time, and modern C++ now has object-oriented, generic, and functional features in addition to facilities for low-level memory manipulation. It is almost always implemented as a compiled language, and many vendors provide C++ compilers, including the Free Software Foundation, LLVM, Microsoft, Intel, Oracle, and IBM, so it is available on many platforms.[9]

C++ was designed with a bias toward system programming and embedded, resource-constrained software and large systems, with performance, efficiency, and flexibility of use as its design highlights.[10] C++ has also been found useful in many other contexts, with key strengths being software infrastructure and resource-constrained applications,[10] including desktop applications, video games, servers (e.g. e-commerce, Web search, or SQL servers), and performance-critical applications (e.g. telephone switches or space probes).[11]

C++ is standardized by the International Organization for Standardization (ISO), with the latest standard version ratified and published by ISO in December 2017 as *ISO/IEC 14882:2017* (informally known as C++17).[12] The C++ programming language was initially standardized in 1998 as *ISO/IEC 14882:1998*, which was then amended by the C++03, C++11 and C++14 standards. The current C++17 standard supersedes these with new features and an enlarged standard library. Before the initial standardization in 1998, C++ was developed by Danish computer scientist Bjarne Stroustrup at Bell Labs since 1979 as an extension of the C language; he wanted an efficient and flexible language similar to C that also provided high-level features for program organization.[13] Since 2012, C++ is on a three-year release schedule,[14] with C++20 the next planned standard (and then C++23).[15]

Contents

C++

The C++ logo endorsed by Standard C++

Paradigms	Multi-paradigm: procedural, functional, object-oriented, generic, modular
Family	C
Designed by	Bjarne Stroustrup
Developer	ISO/IEC JTC1 (Joint Technical Committee 1) / SC22 (Subcommittee 22) / WG21 (Working Group 21)
First appeared	1985
Stable release	C++17 (ISO/IEC 14882:2017) / 1 December 2017
Preview release	C++20
Typing discipline	Static, nominative, partially inferred
OS	Most major
Filename extensions	.C, .cc, .cpp, .cxx, .c++, .h, .hh,

	.hpp, .hxx, .h++
Website	isocpp.org (https://isocpp.org/)
Major implementations	
GCC, LLVM Clang, Microsoft Visual C++, Embarcadero C++Builder, Intel C++ Compiler, IBM XL C++, EDG	
Influenced by	
Ada,[1] ALGOL 68, C, CLU,[1] ML, Mesa,[1] Modula-2,[1] Simula, Smalltalk[1]	
Influenced	
Ada 95, C#,[2] C99, Chapel,[3] Clojure,[4] D, Java,[5] JS++,[6] Lua, Nim,[7] Perl, PHP, Python,[8] Rust, Seed7	
C++ Programming at Wikibooks	

History

In 1979, Bjarne Stroustrup, a Danish computer scientist, began work on "C with Classes", the predecessor to C++.[16] The motivation for creating a new language originated from Stroustrup's experience in programming for his PhD thesis. Stroustrup found that Simula had features that were very helpful for large software development, but the language was too slow for practical use, while BCPL was fast but too low-level to be suitable for large software development. When Stroustrup started working in AT&T Bell Labs, he had the problem of analyzing the UNIX kernel with respect to distributed computing. Remembering his Ph.D. experience, Stroustrup set out to enhance the C language with Simula-like features.[17] C was chosen because it was general-purpose, fast, portable and widely used. As well as C and Simula's influences, other languages also influenced this new language, including ALGOL 68, Ada, CLU and ML.

Bjarne Stroustrup, the creator of C++, in his AT&T New Jersey office c. 2000

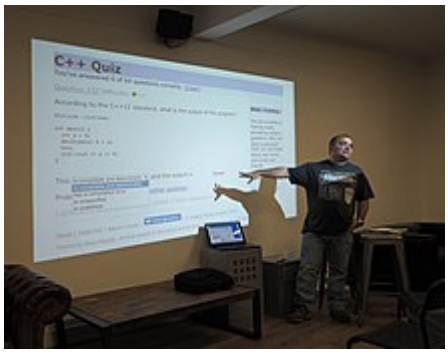
Initially, Stroustrup's "C with Classes" added features to the C compiler, Cpre, including classes, derived classes, strong typing, inlining and default arguments.[18]

In 1982, Stroustrup started to develop a successor to C with Classes, which he named "C++" (`++` being the increment operator in C) after going through several other names. New features were added, including virtual functions, function name and operator overloading, references, constants, type-safe free-store memory allocation (new/delete), improved type checking, and BCPL style single-line comments with two forward slashes (`//`). Furthermore, Stroustrup developed a new, standalone compiler for C++, Cfront.

In 1984, Stroustrup implemented the first stream input/output library. The idea of providing an output operator rather than a named output function was suggested by Doug McIlroy[1] (who had previously suggested Unix pipes).

In 1985, the first edition of *The C++ Programming Language* was released, which became the definitive reference for the language, as there was not yet an official standard.[19] The first commercial implementation of C++ was released in October of the same year.[16]

In 1989, C++ 2.0 was released, followed by the updated second edition of *The C++ Programming Language* in 1991.[20] New features in 2.0 included multiple inheritance, abstract classes, static member functions, const member functions, and protected members. In 1990, *The Annotated C++ Reference Manual* was published. This work became the basis for the future standard. Later feature additions included templates, exceptions, namespaces, new casts, and a Boolean type.



A quiz on C++11 features being given in Paris in 2015

In 1998, C++98 was released, standardizing the language, and a minor update (C++03) was released in 2003.

After C++98, C++ evolved relatively slowly until, in 2011, the C++11 standard was released, adding numerous new features, enlarging the standard library further, and providing more facilities to C++ programmers. After a minor C++14 update released in December 2014, various new additions were introduced in C++17.[21] The C++20 standard became technically finalized in February 2020,[22] and a draft was approved on 4th September 2020; it is expected to be published by the end of 2020.[23][24]

As of 2019, C++ is now the fourth most popular programming language, behind Java, C, and Python.[25][26]

On January 3, 2018, Stroustrup was announced as the 2018 winner of the Charles Stark Draper Prize for Engineering, "for conceptualizing and developing the C++ programming language".[27]

Etymology

According to Stroustrup, "the name signifies the evolutionary nature of the changes from C".[28] This name is credited to Rick Mascitti (mid-1983)[18] and was first used in December 1983. When Mascitti was questioned informally in 1992 about the naming, he indicated that it was given in a tongue-in-cheek spirit. The name comes from C's `++` operator (which increments the value of a variable) and a common naming convention of using "+" to indicate an enhanced computer program.

During C++'s development period, the language had been referred to as "new C" and "C with Classes"[18][29] before acquiring its final name.

Philosophy

Throughout C++'s life, its development and evolution has been guided by a set of principles:[17]

- It must be driven by actual problems and its features should be immediately useful in real world programs.
- Every feature should be implementable (with a reasonably obvious way to do so).
- Programmers should be free to pick their own programming style, and that style should be fully supported by C++.
- Allowing a useful feature is more important than preventing every possible misuse of C++.
- It should provide facilities for organising programs into separate, well-defined parts, and provide facilities for combining separately developed parts.
- No implicit violations of the type system (but allow explicit violations; that is, those explicitly requested by the programmer).
- User-created types need to have the same support and performance as built-in types.
- Unused features should not negatively impact created executables (e.g. in lower performance).
- There should be no language beneath C++ (except assembly language).
- C++ should work alongside other existing programming languages, rather than fostering its own separate and incompatible programming environment.
- If the programmer's intent is unknown, allow the programmer to specify it by providing manual control.

Standardization

Scene during the C++ Standards Committee meeting in Stockholm in 1996

C++ standards

Year	C++ Standard	Informal name
1998	ISO/IEC 14882:1998[30]	C++98
2003	ISO/IEC 14882:2003[31]	C++03
2011	ISO/IEC 14882:2011[32]	C++11, C++0x
2014	ISO/IEC 14882:2014[33]	C++14, C++1y
2017	ISO/IEC 14882:2017[12]	C++17, C++1z
2020	to be determined	C++20,[21] C++2a

C++ is standardized by an ISO working group known as JTC1/SC22/WG21. So far, it has published five revisions of the C++ standard and is currently working on the next revision, C++20.

In 1998, the ISO working group standardized C++ for the first time as *ISO/IEC 14882:1998*, which is informally known as *C++98*. In 2003, it published a new version of the C++ standard called *ISO/IEC 14882:2003*, which fixed problems identified in C++98.

The next major revision of the standard was informally referred to as "C++0x", but it was not released until 2011.[34] C++11 (14882:2011) included many additions to both the core language and the standard library.[32]

In 2014, C++14 (also known as C++1y) was released as a small extension to C++11, featuring mainly bug fixes and small improvements.[35] The Draft International Standard ballot procedures completed in mid-August 2014.[36]

After C++14, a major revision C++17, informally known as C++1z, was completed by the ISO C++ Committee in mid July 2017 and was approved and published in December 2017.[37]

As part of the standardization process, ISO also publishes technical reports and specifications:

- ISO/IEC TR 18015:2006[38] on the use of C++ in embedded systems and on performance implications of C++ language and library features,
- ISO/IEC TR 19768:2007[39] (also known as the C++ Technical Report 1) on library extensions mostly integrated into C++11,
- ISO/IEC TR 29124:2010[40] on special mathematical functions,
- ISO/IEC TR 24733:2011[41] on decimal floating point arithmetic,
- ISO/IEC TS 18822:2015[42] on the standard filesystem library,
- ISO/IEC TS 19570:2015[43] on parallel versions of the standard library algorithms,
- ISO/IEC TS 19841:2015[44] on software transactional memory,
- ISO/IEC TS 19568:2015[45] on a new set of library extensions, some of which are already integrated into C++17,
- ISO/IEC TS 19217:2015[46] on the C++ concepts, integrated into C++20
- ISO/IEC TS 19571:2016[47] on the library extensions for concurrency
- ISO/IEC TS 19568:2017[48] on a new set of general-purpose library extensions
- ISO/IEC TS 21425:2017[49] on the library extensions for ranges, integrated into C++20
- ISO/IEC TS 22277:2017[50] on coroutines
- ISO/IEC TS 19216:2018[51] on the networking library
- ISO/IEC TS 21544:2018[52] on modules
- ISO/IEC TS 19570:2018[53] on a new set of library extensions for parallelism

More technical specifications are in development and pending approval, including static reflection.[54]

Language

The C++ language has two main components: a direct mapping of hardware features provided primarily by the C subset, and zero-overhead abstractions based on those mappings. Stroustrup describes C++ as "a light-weight abstraction programming language [designed] for building and using efficient and elegant abstractions";[10] and "offering both hardware access and abstraction is the basis of C++. Doing it efficiently is what distinguishes it from other languages."[55]

C++ inherits most of C's syntax. The following is Bjarne Stroustrup's version of the Hello world program that uses the C++ Standard Library stream facility to write a message to standard output:[56][57]

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Hello, world!\n";
6 }
```

Object storage

As in C, C++ supports four types of memory management: static storage duration objects, thread storage duration objects, automatic storage duration objects, and dynamic storage duration objects.[58]

Static storage duration objects

Static storage duration objects are created before `main()` is entered (see exceptions below) and destroyed in reverse order of creation after `main()` exits. The exact order of creation is not specified by the standard (though there are some rules defined below) to allow implementations some freedom in how to organize their implementation. More formally, objects of this type have a lifespan that "shall last for the duration of the program".[59]

Static storage duration objects are initialized in two phases. First, "static initialization" is performed, and only *after* all static initialization is performed, "dynamic initialization" is performed. In static initialization, all objects are first initialized with zeros; after that, all objects that have a constant initialization phase are initialized with the constant expression (i.e. variables initialized with a literal or `constexpr`). Though it is not specified in the standard, the static initialization phase can be completed at compile time and saved in the data partition of the executable. Dynamic initialization involves all object initialization done via a constructor or function call (unless the function is marked with `constexpr`, in C++11). The dynamic initialization order is defined as the order of declaration within the compilation unit (i.e. the same file). No guarantees are provided about the order of initialization between compilation units.

Thread storage duration objects

Variables of this type are very similar to static storage duration objects. The main difference is the creation time is just prior to thread creation and destruction is done after the thread has been joined.[60]

Automatic storage duration objects

The most common variable types in C++ are local variables inside a function or block, and temporary variables.[61] The common feature about automatic variables is that they have a lifetime that is limited to the scope of the variable. They are created and potentially initialized at the point of declaration (see below for details) and destroyed in the *reverse* order of creation when the scope is left. This is implemented by allocation on the stack.

Local variables are created as the point of execution passes the declaration point. If the variable has a constructor or initializer this is used to define the initial state of the object. Local variables are destroyed when the local block or function that they are declared in is closed. C++ destructors for local variables are called at the end of the object lifetime, allowing a discipline for automatic resource management termed RAII, which is widely used in C++.

Member variables are created when the parent object is created. Array members are initialized from 0 to the last member of the array in order. Member variables are destroyed when the parent object is destroyed in the reverse order of creation. i.e. If the parent is an "automatic object" then it will be destroyed when it goes out of scope which triggers the destruction of all its members.

Temporary variables are created as the result of expression evaluation and are destroyed when the statement containing the expression has been fully evaluated (usually at the `;` at the end of a statement).

Dynamic storage duration objects

These objects have a dynamic lifespan and can be created directly with a call to `new` and destroyed explicitly with a call to `delete`.[62] C++ also supports `malloc` and `free`, from C, but these are not compatible with `new` and `delete`. Use of `new` returns an address to the allocated memory. The C++ Core Guidelines advise

against using `new` directly for creating dynamic objects in favor of smart pointers through `make_unique<T>` for single ownership and `make_shared<T>` for reference-counted multiple ownership,[63] which were introduced in C++11.

Templates

C++ templates enable generic programming. C++ supports function, class, alias, and variable templates. Templates may be parameterized by types, compile-time constants, and other templates. Templates are implemented by *instantiation* at compile-time. To instantiate a template, compilers substitute specific arguments for a template's parameters to generate a concrete function or class instance. Some substitutions are not possible; these are eliminated by an overload resolution policy described by the phrase "Substitution failure is not an error" (SFINAE). Templates are a powerful tool that can be used for generic programming, template metaprogramming, and code optimization, but this power implies a cost. Template use may increase code size, because each template instantiation produces a copy of the template code: one for each set of template arguments, however, this is the same or smaller amount of code that would be generated if the code was written by hand.[64] This is in contrast to run-time generics seen in other languages (e.g., Java) where at compile-time the type is erased and a single template body is preserved.

Templates are different from macros: while both of these compile-time language features enable conditional compilation, templates are not restricted to lexical substitution. Templates are aware of the semantics and type system of their companion language, as well as all compile-time type definitions, and can perform high-level operations including programmatic flow control based on evaluation of strictly type-checked parameters. Macros are capable of conditional control over compilation based on predetermined criteria, but cannot instantiate new types, recurse, or perform type evaluation and in effect are limited to pre-compilation text-substitution and text-inclusion/exclusion. In other words, macros can control compilation flow based on pre-defined symbols but cannot, unlike templates, independently instantiate new symbols. Templates are a tool for static polymorphism (see below) and generic programming.

In addition, templates are a compile time mechanism in C++ that is Turing-complete, meaning that any computation expressible by a computer program can be computed, in some form, by a template metaprogram prior to runtime.

In summary, a template is a compile-time parameterized function or class written without knowledge of the specific arguments used to instantiate it. After instantiation, the resulting code is equivalent to code written specifically for the passed arguments. In this manner, templates provide a way to decouple generic, broadly applicable aspects of functions and classes (encoded in templates) from specific aspects (encoded in template parameters) without sacrificing performance due to abstraction.

Objects

C++ introduces object-oriented programming (OOP) features to C. It offers classes, which provide the four features commonly present in OOP (and some non-OOP) languages: abstraction, encapsulation, inheritance, and polymorphism. One distinguishing feature of C++ classes compared to classes in other programming languages is support for deterministic destructors, which in turn provide support for the Resource Acquisition is Initialization (RAII) concept.

Encapsulation

Encapsulation is the hiding of information to ensure that data structures and operators are used as intended and to make the usage model more obvious to the developer. C++ provides the ability to define classes and functions as its primary encapsulation mechanisms. Within a class, members can be declared as either public, protected, or private to explicitly enforce encapsulation. A public member of the class is accessible to any function. A private member is accessible only to functions that are members of that class and to functions and classes explicitly granted access permission by the class ("friends"). A protected member is accessible to members of classes that inherit from the class in addition to the class itself and any friends.

The object-oriented principle ensures the encapsulation of all and only the functions that access the internal representation of a type. C++ supports this principle via member functions and friend functions, but it does not enforce it. Programmers can declare parts or all of the representation of a type to be public, and they are allowed to make public entities not part of the representation of a type. Therefore, C++ supports not just object-oriented programming, but other decomposition paradigms such as modular programming.

It is generally considered good practice to make all data private or protected, and to make public only those functions that are part of a minimal interface for users of the class. This can hide the details of data implementation, allowing the designer to later fundamentally change the implementation without changing the interface in any way.[65][66]

Inheritance

Inheritance allows one data type to acquire properties of other data types. Inheritance from a base class may be declared as public, protected, or private. This access specifier determines whether unrelated and derived classes can access the inherited public and protected members of the base class. Only public inheritance corresponds to what is usually meant by "inheritance". The other two forms are much less frequently used. If the access specifier is omitted, a "class" inherits privately, while a "struct" inherits publicly. Base classes may be declared as virtual; this is called virtual inheritance. Virtual inheritance ensures that only one instance of a base class exists in the inheritance graph, avoiding some of the ambiguity problems of multiple inheritance.

Multiple inheritance is a C++ feature not found in most other languages, allowing a class to be derived from more than one base class; this allows for more elaborate inheritance relationships. For example, a "Flying Cat" class can inherit from both "Cat" and "Flying Mammal". Some other languages, such as C# or Java, accomplish something similar (although more limited) by allowing inheritance of multiple interfaces while restricting the number of base classes to one (interfaces, unlike classes, provide only declarations of member functions, no implementation or member data). An interface as in C# and Java can be defined in C++ as a class containing only pure virtual functions, often known as an abstract base class or "ABC". The member functions of such an abstract base class are normally explicitly defined in the derived class, not inherited implicitly. C++ virtual inheritance exhibits an ambiguity resolution feature called dominance.

Operators and operator overloading

C++ provides more than 35 operators, covering basic arithmetic, bit manipulation, indirection, comparisons, logical operations and others. Almost all operators can be overloaded for user-defined types, with a few notable exceptions such as member access (`.` and `.*`) as well as the conditional operator. The rich set of overloadable operators is central to making user-defined types in C++ seem like built-in types.

Overloadable operators are also an essential part of many advanced C++ programming techniques, such as smart pointers. Overloading an operator does not change the precedence of calculations involving the operator, nor does it change the number of operands that the operator uses (any operand may however be ignored by the operator, though it will be evaluated prior to execution). Overloaded "&&" and "`||`" operators lose their short-circuit evaluation property.

Polymorphism

Polymorphism enables one common interface for many implementations, and for objects to act differently under different circumstances.

C++ supports several kinds of *static* (resolved at compile-time) and *dynamic* (resolved at run-time) polymorphisms, supported by the language features described above. Compile-time polymorphism does not allow for certain run-time decisions, while runtime polymorphism typically incurs a performance penalty.

Operators that cannot be overloaded

Operator	Symbol
Scope resolution operator	`::`
Conditional operator	`?:`
dot operator	`.`
Member selection operator	`.*`
"sizeof" operator	`sizeof`
"typeid" operator	`typeid`

Static polymorphism

Function overloading allows programs to declare multiple functions having the same name but with different arguments (i.e. *ad hoc* polymorphism). The functions are distinguished by the number or types of their formal parameters. Thus, the same function name can refer to different functions depending on the context in which it is used. The type returned by the function is not used to distinguish overloaded functions and would result in a compile-time error message.

When declaring a function, a programmer can specify for one or more parameters a default value. Doing so allows the parameters with defaults to optionally be omitted when the function is called, in which case the default arguments will be used. When a function is called with fewer arguments than there are declared parameters, explicit arguments are matched to parameters in left-to-right order, with any unmatched parameters at the end of the parameter list being assigned their default arguments. In many cases, specifying default arguments in a single function declaration is preferable to providing overloaded function definitions with different numbers of parameters.

Templates in C++ provide a sophisticated mechanism for writing generic, polymorphic code (i.e. parametric polymorphism). In particular, through the curiously recurring template pattern, it's possible to implement a form of static polymorphism that closely mimics the syntax for overriding virtual functions. Because C++ templates are type-aware and Turing-complete, they can also be used to let the compiler resolve recursive conditionals and generate substantial programs through template metaprogramming. Contrary to some opinion, template code will not generate a bulk code after compilation with the proper compiler settings.[64]

Dynamic polymorphism

Inheritance

Variable pointers and references to a base class type in C++ can also refer to objects of any derived classes of that type. This allows arrays and other kinds of containers to hold pointers to objects of differing types (references cannot be directly held in containers). This enables dynamic (run-time) polymorphism, where the referred objects can behave differently, depending on their (actual, derived) types.

C++ also provides the `dynamic_cast` operator, which allows code to safely attempt conversion of an object, via a base reference/pointer, to a more derived type: *downcasting*. The *attempt* is necessary as often one does not know which derived type is referenced. (*Upcasting*, conversion to a more general type, can always be checked/performed at compile-time via `static_cast`, as ancestral classes are specified in the derived class's interface, visible to all callers.) `dynamic_cast` relies on run-time type information (RTTI), metadata

in the program that enables differentiating types and their relationships. If a `dynamic_cast` to a pointer fails, the result is the `nullptr` constant, whereas if the destination is a reference (which cannot be null), the cast throws an exception. Objects *known* to be of a certain derived type can be cast to that with `static_cast`, bypassing RTTI and the safe runtime type-checking of `dynamic_cast`, so this should be used only if the programmer is very confident the cast is, and will always be, valid.

Virtual member functions

Ordinarily, when a function in a derived class overrides a function in a base class, the function to call is determined by the type of the object. A given function is overridden when there exists no difference in the number or type of parameters between two or more definitions of that function. Hence, at compile time, it may not be possible to determine the type of the object and therefore the correct function to call, given only a base class pointer; the decision is therefore put off until runtime. This is called dynamic dispatch. Virtual member functions or *methods*[67] allow the most specific implementation of the function to be called, according to the actual run-time type of the object. In C++ implementations, this is commonly done using virtual function tables. If the object type is known, this may be bypassed by prepending a fully qualified class name before the function call, but in general calls to virtual functions are resolved at run time.

In addition to standard member functions, operator overloads and destructors can be virtual. As a rule of thumb, if any function in the class is virtual, the destructor should be as well. As the type of an object at its creation is known at compile time, constructors, and by extension copy constructors, cannot be virtual. Nonetheless a situation may arise where a copy of an object needs to be created when a pointer to a derived object is passed as a pointer to a base object. In such a case, a common solution is to create a `clone()` (or similar) virtual function that creates and returns a copy of the derived class when called.

A member function can also be made "pure virtual" by appending it with `= 0` after the closing parenthesis and before the semicolon. A class containing a pure virtual function is called an *abstract class*. Objects cannot be created from an abstract class; they can only be derived from. Any derived class inherits the virtual function as pure and must provide a non-pure definition of it (and all other pure virtual functions) before objects of the derived class can be created. A program that attempts to create an object of a class with a pure virtual member function or inherited pure virtual member function is ill-formed.

Lambda expressions

C++ provides support for anonymous functions, also known as lambda expressions, with the following form:

```
[capture](parameters) -> return_type { function_body }
```

If the lambda takes no parameters, the () can be omitted, that is,

```
[capture] -> return_type { function_body }
```

Also, the return type of a lambda expression can be automatically inferred, if possible, e.g.:

```
[](int x, int y) { return x + y; } // inferred
[](int x, int y) -> int { return x + y; } // explicit
```

The `[capture]` list supports the definition of closures. Such lambda expressions are defined in the standard as syntactic sugar for an unnamed function object.

Exception handling

Exception handling is used to communicate the existence of a runtime problem or error from where it was detected to where the issue can be handled.[68] It permits this to be done in a uniform manner and separately from the main code, while detecting all errors.[69] Should an error occur, an exception is thrown (raised), which is then caught by the nearest suitable exception handler. The exception causes the current scope to be exited, and also each outer scope (propagation) until a suitable handler is found, calling in turn the destructors of any objects in these exited scopes.[70] At the same time, an exception is presented as an object carrying the data about the detected problem.[71]

Some C++ style guides, such as Google's,[72] LLVM's,[73] and Qt's[74] forbid the usage of exceptions.

The exception-causing code is placed inside a `try` block. The exceptions are handled in separate `catch` blocks (the handlers); each `try` block can have multiple exception handlers, as it is visible in the example below.[75]

```
 1 #include <iostream>
 2 #include <vector>
 3 #include <stdexcept>
 4
 5 int main() {
 6     try {
 7         std::vector<int> vec{3, 4, 3, 1};
 8         int i{vec.at(4)}; // Throws an exception, std::out_of_range (indexing for vec is from
0-3 not 1-4)
 9     }
10     // An exception handler, catches std::out_of_range, which is thrown by vec.at(4)
11     catch (std::out_of_range &e) {
12         std::cerr << "Accessing a non-existent element: " << e.what() << '\n';
13     }
14     // To catch any other standard library exceptions (they derive from std::exception)
15     catch (std::exception &e) {
16         std::cerr << "Exception thrown: " << e.what() << '\n';
17     }
18     // Catch any unrecognised exceptions (i.e. those which don't derive from std::exception)
19     catch (...) {
20         std::cerr << "Some fatal error\n";
21     }
22 }
```

It is also possible to raise exceptions purposefully, using the `throw` keyword; these exceptions are handled in the usual way. In some cases, exceptions cannot be used due to technical reasons. One such example is a critical component of an embedded system, where every operation must be guaranteed to complete within a specified amount of time. This cannot be determined with exceptions as no tools exist to determine the maximum time required for an exception to be handled.[76]

Unlike signal handling, in which the handling function is called from the point of failure, exception handling exits the current scope before the catch block is entered, which may be located in the current function or any of the previous function calls currently on the stack.

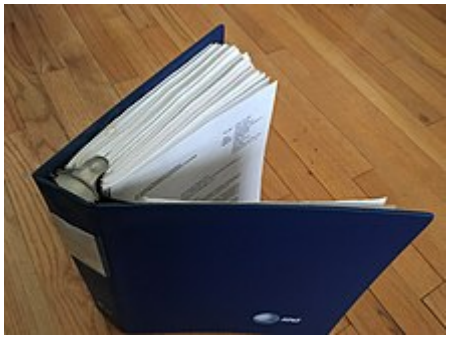
Standard library

The C++ standard consists of two parts: the core language and the standard library. C++ programmers expect the latter on every major implementation of C++; it includes aggregate types (vectors, lists, maps, sets, queues, stacks, arrays, tuples), algorithms (find, for_each, binary_search, random_shuffle, etc.), input/output facilities (iostream, for reading from and writing to the console and files), filesystem library, localisation support, smart pointers for automatic memory management, regular expression support, multi-threading library, atomics support (allowing a variable to be read or written to by at most one thread at a time without any external synchronisation), time utilities (measurement, getting current time, etc.), a system for converting error reporting

that doesn't use C++ exceptions into C++ exceptions, a random number generator and a slightly modified version of the C standard library (to make it comply with the C++ type system).

A large part of the C++ library is based on the Standard Template Library (STL). Useful tools provided by the STL include containers as the collections of objects (such as vectors and lists), iterators that provide array-like access to containers, and algorithms that perform operations such as searching and sorting.

Furthermore, (multi)maps (associative arrays) and (multi)sets are provided, all of which export compatible interfaces. Therefore, using templates it is possible to write generic algorithms that work with any container or on any sequence defined by iterators. As in C, the features of the library are accessed by using the `#include` directive to include a standard header. The C++ Standard Library provides 105 standard headers, of which 27 are deprecated.



The draft "Working Paper" standard that became approved as C++98; half of its size was devoted to the C++ Standard Library

The standard incorporates the STL that was originally designed by Alexander Stepanov, who experimented with generic algorithms and containers for many years. When he started with C++, he finally found a language where it was possible to create generic algorithms (e.g., STL sort) that perform even better than, for example, the C standard library qsort, thanks to C++ features like using inlining and compile-time binding instead of function pointers. The standard does not refer to it as "STL", as it is merely a part of the standard library, but the term is still widely used to distinguish it from the rest of the standard library (input/output streams, internationalization, diagnostics, the C library subset, etc.).[77]

Most C++ compilers, and all major ones, provide a standards-conforming implementation of the C++ standard library.

C++ Core Guidelines

The C++ Core Guidelines[78] are an initiative led by Bjarne Stroustrup, the inventor of C++, and Herb Sutter, the convener and chair of the C++ ISO Working Group, to help programmers write 'Modern C++' by using best practices for the language standards C++14 and newer, and to help developers of compilers and static checking tools to create rules for catching bad programming practices.

The main aim is to efficiently and consistently write type and resource safe C++.

The Core Guidelines were announced[79] in the opening keynote at CPPCon 2015.

The Guidelines are accompanied by the Guideline Support Library (GSL),[80] a header only library of types and functions to implement the Core Guidelines and static checker tools for enforcing Guideline rules.[81]

Compatibility

To give compiler vendors greater freedom, the C++ standards committee decided not to dictate the implementation of name mangling, exception handling, and other implementation-specific features. The downside of this decision is that object code produced by different compilers is expected to be incompatible. There were, however, attempts to standardize compilers for particular machines or operating systems (for example C++ ABI),[82] though they seem to be largely abandoned now.

With C

C++ is often considered to be a superset of C but this is not strictly true.[83] Most C code can easily be made to compile correctly in C++ but there are a few differences that cause some valid C code to be invalid or behave differently in C++. For example, C allows implicit conversion from `void*` to other pointer types but C++ does not (for type safety reasons). Also, C++ defines many new keywords, such as `new` and `class`, which may be used as identifiers (for example, variable names) in a C program.

Some incompatibilities have been removed by the 1999 revision of the C standard (C99), which now supports C++ features such as line comments (`//`) and declarations mixed with code. On the other hand, C99 introduced a number of new features that C++ did not support that were incompatible or redundant in C++, such as variable-length arrays, native complex-number types (however, the `std::complex` class in the C++ standard library provides similar functionality, although not code-compatible), designated initializers, compound literals, and the `restrict` keyword.[84] Some of the C99-introduced features were included in the subsequent version of the C++ standard, C++11 (out of those which were not redundant).[85][86][87] However, the C++11 standard introduces new incompatibilities, such as disallowing assignment of a string literal to a character pointer, which remains valid C.

To intermix C and C++ code, any function declaration or definition that is to be called from/used both in C and C++ must be declared with C linkage by placing it within an `extern "C" {/*...*/}` block. Such a function may not rely on features depending on name mangling (i.e., function overloading).

Criticism

Despite its widespread adoption, some notable programmers have criticized the C++ language, including Linus Torvalds,[88] Richard Stallman,[89] Joshua Bloch, Ken Thompson,[90][91][92] and Donald Knuth.[93][94]

One of the most often criticised points of C++ is its perceived complexity as a language, with the criticism that a large number of non-orthogonal features in practice necessitates restricting code to subset of C++, thus eschewing the readability benefits of common style and idioms. As expressed by Joshua Bloch:

> I think C++ was pushed well beyond its complexity threshold, and yet there are a lot of people programming it. But what you do is you force people to subset it. So almost every shop that I know of that uses C++ says, “Yes, we’re using C++ but we’re not doing multiple-implementation inheritance and we’re not using operator overloading.” There are just a bunch of features that you’re not going to use because the complexity of the resulting code is too high. And I don’t think it’s good when you have to start doing that. You lose this programmer portability where everyone can read everyone else’s code, which I think is such a good thing.

Donald Knuth (1993, commenting on pre-standardized C++), who said of Edsger Dijkstra that "to think of programming in C++" "would make him physically ill":[93][94]

> The problem that I have with them today is that... C++ is too complicated. At the moment, it's impossible for me to write portable code that I believe would work on lots of different systems, unless I avoid all exotic features. Whenever the C++ language designers had two competing ideas as to how they should solve some problem, they said "OK, we'll do them both". So the language is too baroque for my taste.

Ken Thompson, who was a colleague of Stroustrup at Bell Labs, gives his assessment:[91][92]

> It certainly has its good points. But by and large I think it's a bad language. It does a lot of things half well and it's just a garbage heap of ideas that are mutually exclusive. Everybody I know, whether it's personal or corporate, selects a subset and these subsets are different. So it's not a good language to transport an algorithm—to say, "I wrote it; here, take it." It's way too big, way too complex. And it's obviously built by a committee. Stroustrup campaigned for years and years and years, way beyond any sort of technical contributions he made to the language, to get it adopted and used. And he sort of ran all the standards committees with a whip and a chair. And he said "no" to no one. He put every feature in that language that ever existed. It wasn't cleanly designed—it was just the union of everything that came along. And I think it suffered drastically from that.

However Brian Kernighan, also a colleague at Bell Labs, disputes this assessment:[95]

> C++ has been enormously influential. ... Lots of people say C++ is too big and too complicated etc. etc. but in fact it is a very powerful language and pretty much everything that is in there is there for a really sound reason: it is not somebody doing random invention, it is actually people trying to solve real world problems. Now a lot of the programs that we take for granted today, that we just use, are C++ programs.

Stroustrup himself comments that C++ semantics are much cleaner than its syntax: "within C++, there is a much smaller and cleaner language struggling to get out".[96]

Other complaints may include a lack of reflection or garbage collection, long compilation times, perceived feature creep,[97] and verbose error messages, particularly from template metaprogramming.[98]

See also

- Comparison of programming languages
- List of C++ compilers
- Outline of C++
- Category:C++ libraries

References

1. Bjarne Stroustrup. "A history of C++: 1979-1991". doi:10.1145/234286.1057836 (https://doi.org/10.1145%2F234286.1057836).
2. Naugler, David (May 2007). "C# 2.0 for C++ and Java programmer: conference workshop". *Journal of Computing Sciences in Colleges*. **22** (5). "Although C# has been strongly influenced by Java it has also been strongly influenced by C++ and is best viewed as a descendant of both C++ and Java."
3. "Chapel spec (Acknowledgements)" (https://chapel-lang.org/spec/spec-0.98.pdf) (PDF). Cray Inc. 1 October 2015. Retrieved 14 January 2016.
4. "Rich Hickey Q&A by Michael Fogus" (https://web.archive.org/web/20170111184835/http://www.codequarterly.com/2011/rich-hickey/). Archived from the original (http://www.codequarterly.com/2011/rich-hickey/) on 11 January 2017. Retrieved 11 January 2017.

5. Harry. H. Chaudhary (28 July 2014). "Cracking The Java Programming Interview :: 2000+ Java Interview Que/Ans" (https://books.google.com/books?id=0rUtBAAAQBAJ&pg=PA133). Retrieved 29 May 2016.
6. Roger Poon (1 May 2017). "Scaling JS++: Abstraction, Performance, and Readability" (https://www.onux.com/jspp/blog/scaling-jspp-abstraction-performance-and-readability/). Retrieved 21 April 2020.
7. "FAQ Nim Programming Language" (https://nim-lang.org/faq.html). Retrieved 21 April 2020.
8. "9. Classes — Python 3.6.4 documentation" (https://docs.python.org/tutorial/classes.html). *docs.python.org*. Retrieved 9 January 2018.
9. Stroustrup, Bjarne (1997). "1". *The C++ Programming Language* (https://archive.org/details/cprogramminglang00stro_0) (Third ed.). ISBN 0-201-88954-4. OCLC 59193992 (https://www.worldcat.org/oclc/59193992).
10. Stroustrup, B. (6 May 2014). "Lecture:The essence of C++. University of Edinburgh" (https://www.youtube.com/watch?v=86xWVb4XIyE). Retrieved 12 June 2015.
11. Stroustrup, Bjarne (17 February 2014). "C++ Applications" (http://www.stroustrup.com/applications.html). *stroustrup.com*. Retrieved 5 May 2014.
12. "ISO/IEC 14882:2017" (https://www.iso.org/standard/68564.html). International Organization for Standardization.
13. "Bjarne Stroustrup's Homepage" (http://www.stroustrup.com). *www.stroustrup.com*.
14. "C++ IS schedule" (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1000r4.pdf) (PDF).
15. "C++; Where it's heading" (https://dzone.com/articles/c-where-is-it-heading-and-what-are-the-new-feature).
16. Stroustrup, Bjarne (7 March 2010). "Bjarne Stroustrup's FAQ: When was C++ invented?" (http://www.stroustrup.com/bs_faq.html#invention). *stroustrup.com*. Retrieved 16 September 2010.
17. Stroustrup, Bjarne. "Evolving a language in and for the real world: C++ 1991-2006" (http://stroustrup.com/hopl-almost-final.pdf) (PDF).
18. Stroustrup, Bjarne. "A History of C ++ : 1979− 1991" (http://www.stroustrup.com/hopl2.pdf) (PDF).
19. Stroustrup, Bjarne. "The C++ Programming Language" (http://www.stroustrup.com/1st.html) (First ed.). Retrieved 16 September 2010.
20. Stroustrup, Bjarne. "The C++ Programming Language" (http://www.stroustrup.com/2nd.html) (Second ed.). Retrieved 16 September 2010.
21. https://herbsutter.com/2016/06/30/trip-report-summer-iso-c-standards-meeting-oulu/ "the next standard after C++17 will be C++20"
22. Dusíková, Hana (6 November 2019). "N4817: 2020 Prague Meeting Invitation and Information" (http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/n4817.pdf) (PDF). Retrieved 13 February 2020.
23. "Current Status" (https://isocpp.org/std/status). *isocpp.org*. Retrieved 7 September 2020.
24. "C++20 Approved -- Herb Sutter" (https://isocpp.org/blog/2020/09/cpp20-approved-herb-sutter). *isocpp.org*. Retrieved 8 September 2020.
25. "Latest news." TIOBE Index | TIOBE - The Software Quality Company. N.p., n.d. Web. 5 June 2017.
26. Krill, Paul. "Java, C, C face growing competition in popularity." InfoWorld. InfoWorld, 10 February 2017. Web. 5 June 2017.
27. https://www.nae.edu/177355.aspx "Computer Science Pioneer Bjarne Stroustrup to Receive the 2018 Charles Stark Draper Prize for Engineering"
28. "Bjarne Stroustrup's FAQ – Where did the name "C++" come from?" (http://www.stroustrup.com/bs_faq.html#name). Retrieved 16 January 2008.

29. "C For C++ Programmers" (https://web.archive.org/web/20101117003419/http://www.ccs.neu.edu/course/com3620/parent/C-for-Java-C++/c-for-c++-alt.html). Northeastern University. Archived from the original (https://www.ccs.neu.edu/course/com3620/parent/C-for-Java-C++/c-for-c++-alt.html) on 17 November 2010. Retrieved 7 September 2015.
30. "ISO/IEC 14882:1998" (https://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?ics1=35&ics2=60&ics3=&csnumber=25845). International Organization for Standardization.
31. "ISO/IEC 14882:2003" (https://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?ics1=35&ics2=60&ics3=&csnumber=38110). International Organization for Standardization.
32. "ISO/IEC 14882:2011" (https://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?ics1=35&ics2=60&ics3=&csnumber=50372). International Organization for Standardization.
33. "ISO/IEC 14882:2014" (https://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=64029&ICS1=35&ICS2=60). International Organization for Standardization.
34. "We have an international standard: C++0x is unanimously approved" (https://herbsutter.com/2011/08/12/we-have-an-international-standard-c0x-is-unanimously-approved/). *Sutter's Mill*.
35. "The Future of C++" (https://channel9.msdn.com/Events/Build/2012/2-005).
36. "We have C++14! : Standard C++" (https://isocpp.org/blog/2014/08/we-have-cpp14).
37. Trip report: Summer ISO C++ standards meeting (Toronto) (https://herbsutter.com/2017/07/15/trip-report-summer-iso-c-standards-meeting-toronto/)
38. "ISO/IEC TR 18015:2006" (https://www.iso.org/standard/43351.html). International Organization for Standardization.
39. "ISO/IEC TR 19768:2007" (https://www.iso.org/standard/43289.html). International Organization for Standardization.
40. "ISO/IEC TR 29124:2010" (https://www.iso.org/standard/50511.html). International Organization for Standardization.
41. "ISO/IEC TR 24733:2011" (https://www.iso.org/standard/38843.html). International Organization for Standardization.
42. "ISO/IEC TS 18822:2015" (https://www.iso.org/standard/63483.html). International Organization for Standardization.
43. "ISO/IEC TS 19570:2015" (https://www.iso.org/standard/65241.html). International Organization for Standardization.
44. "ISO/IEC TS 19841:2015" (https://www.iso.org/standard/66343.html). International Organization for Standardization.
45. "ISO/IEC TS 19568:2015" (https://www.iso.org/standard/65238.html). International Organization for Standardization.
46. "ISO/IEC TS 19217:2015" (https://www.iso.org/standard/64031.html). International Organization for Standardization.
47. "ISO/IEC TS 19571:2016" (https://www.iso.org/standard/65242.html). International Organization for Standardization.
48. "ISO/IEC TS 19568:2017" (https://www.iso.org/standard/70587.html). International Organization for Standardization.
49. "ISO/IEC TS 21425:2017" (https://www.iso.org/standard/70910.html). International Organization for Standardization.
50. "ISO/IEC TS 22277:2017" (https://www.iso.org/standard/73008.html). International Organization for Standardization.
51. "ISO/IEC TS 19216:2018" (https://www.iso.org/standard/64030.html). International Organization for Standardization.

52. "ISO/IEC TS 21544:2018" (https://www.iso.org/standard/71051.html). International Organization for Standardization.
53. "ISO/IEC TS 19570:2018" (https://www.iso.org/standard/70588.html). International Organization for Standardization.
54. See a list at https://en.cppreference.com/w/cpp/experimental visited 15 February 2019.
55. B. Stroustrup (interviewed by Sergio De Simone) (30 April 2015). "Stroustrup: Thoughts on C++17 - An Interview" (https://www.infoq.com/news/2015/04/stroustrup-cpp17-interview). Retrieved 8 July 2015.
56. Stroustrup, Bjarne (2000). *The C++ Programming Language* (Special ed.). Addison-Wesley. p. 46. ISBN 0-201-70073-5.
57. Stroustrup, Bjarne. "Open issues for The C++ Programming Language (3rd Edition)" (http://www.stroustrup.com/3rd_issues.html). This code is copied directly from Bjarne Stroustrup's errata page (p. 633). He addresses the use of `'\n'` rather than `std::endl`. Also see Can I write "void main()"? (http://www.stroustrup.com/bs_faq2.html#void-main) for an explanation of the implicit `return 0;` in the `main` function. This implicit return is *not* available in other functions.
58. ISO/IEC. *Programming Languages – C++11 Draft (n3797) (https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf) Archived (https://web.archive.org/web/20181002093659/http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf) 2 October 2018 at the Wayback Machine §3.7 Storage duration [basic.stc]*
59. ISO/IEC. *Programming Languages – C++11 Draft (n3797) (https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf) Archived (https://web.archive.org/web/20181002093659/http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf) 2 October 2018 at the Wayback Machine §3.7.1 Static Storage duration [basic.stc.static]*
60. ISO/IEC. *Programming Languages – C++11 Draft (n3797) (https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf) Archived (https://web.archive.org/web/20181002093659/http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf) 2 October 2018 at the Wayback Machine §3.7.2 Thread Storage duration [basic.stc.thread]*
61. ISO/IEC. *Programming Languages – C++11 Draft (n3797) (https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf) Archived (https://web.archive.org/web/20181002093659/http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf) 2 October 2018 at the Wayback Machine §3.7.3 Automatic Storage duration [basic.stc.auto]*
62. ISO/IEC. *Programming Languages – C++11 Draft (n3797) (https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf) Archived (https://web.archive.org/web/20181002093659/http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf) 2 October 2018 at the Wayback Machine §3.7.4 Dynamic Storage duration [basic.stc.dynamic]*
63. "C++ Core Guidelines" (https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#r11-avoid-calling-new-and-delete-explicitly). *isocpp.github.io*. Retrieved 9 February 2020.
64. "Nobody Understands C++: Part 5: Template Code Bloat" (https://articles.emptycrate.com/2008/05/06/nobody_understands_c_part_5_template_code_bloat.html). articles.emptycrate.com/: EmptyCrate Software. Travel. Stuff. 6 May 2008. Retrieved 8 March 2010. "On occasion you will read or hear someone talking about C++ templates causing code bloat. I was thinking about it the other day and thought to myself, "self, if the code does exactly the same thing then the compiled code cannot really be any bigger, can it?" [...] And what about compiled code size? Each were compiled with the command g++ <filename>.cpp -O3. Non-template version: 8140 bytes, template version: 8028 bytes!"
65. Sutter, Herb; Alexandrescu, Andrei (2004). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley.
66. Henricson, Mats; Nyquist, Erik (1997). *Industrial Strength C++* (https://archive.org/details/industrialstreng0000henr). Prentice Hall. ISBN 0-13-120965-5.
67. Stroustrup, Bjarne (2000). *The C++ Programming Language* (Special ed.). Addison-Wesley. p. 310. ISBN 0-201-70073-5. "A virtual member function is sometimes called a *method*."

68. Mycroft, Alan (2013). "C and C++ Exceptions | Templates" (http://www.cl.cam.ac.uk/teaching/1314/CandC++/lecture7.pdf) (PDF). *Cambridge Computer Laboratory - Course Materials 2013-14*. Retrieved 30 August 2016.
69. Stroustrup, Bjarne (2013). *The C++ Programming Language*. Addison Wesley. p. 345. ISBN 9780321563842.
70. Stroustrup, Bjarne (2013). *The C++ Programming Language*. Addison Wesley. pp. 363–365. ISBN 9780321563842.
71. Stroustrup, Bjarne (2013). *The C++ Programming Language*. Addison Wesley. pp. 345, 363. ISBN 9780321563842.
72. "Google C++ Style Guide" (https://google.github.io/styleguide/cppguide.html#Exceptions). Retrieved 25 June 2019.
73. "LLVM Coding Standards" (https://llvm.org/docs/CodingStandards.html#do-not-use-rtti-or-exceptions). *LLVM 9 documentation*. Retrieved 25 June 2019.
74. "Coding Conventions" (https://wiki.qt.io/Coding_Conventions). *Qt Wiki*. Retrieved 26 June 2019.
75. Stroustrup, Bjarne (2013). *The C++ Programming Language*. Addison Wesley. pp. 344, 370. ISBN 9780321563842.
76. Stroustrup, Bjarne (2013). *The C++ Programming Language*. Addison Wesley. p. 349. ISBN 9780321563842.
77. Graziano Lo Russo (2008). "An Interview with A. Stepanov" (http://www.stlport.org/resources/StepanovUSA.html). *stlport.org*. Retrieved 8 October 2015.
78. "C++ Core Guidelines" (https://isocpp.github.io/CppCoreGuidelines).
79. "Bjarne Stroustrup announces C++ Core Guidelines" (https://isocpp.org/blog/2015/09/bjarne-stroustrup-announces-cpp-core-guidelines).
80. "Guidelines Support Library" (https://github.com/Microsoft/GSL).
81. "Use the C++ Core Guidelines checkers" (https://docs.microsoft.com/en-us/cpp/code-quality/using-the-cpp-core-guidelines-checkers?view=vs-2019).
82. "C++ ABI Summary" (https://mentorembedded.github.io/cxx-abi/). 20 March 2001. Retrieved 30 May 2006.
83. "Bjarne Stroustrup's FAQ – Is C a subset of C++?" (http://www.stroustrup.com/bs_faq.html#C-is-subset). Retrieved 5 May 2014.
84. "C9X – The New C Standard" (http://home.datacomm.ch/t_wolf/tw/c/c9x_changes.html). Retrieved 27 December 2008.
85. "C++0x Support in GCC" (https://gcc.gnu.org/projects/cxx0x.html). Retrieved 12 October 2010.
86. "C++0x Core Language Features In VC10: The Table" (https://blogs.msdn.com/b/vcblog/archive/2010/04/06/c-0x-core-language-features-in-vc10-the-table.aspx). Retrieved 12 October 2010.
87. "Clang - C++98, C++11, and C++14 Status" (https://clang.llvm.org/cxx_status.html). Clang.llvm.org. 12 May 2013. Retrieved 10 June 2013.
88. "Re: [RFC] Convert builin-mailinfo.c to use The Better String Library" (https://lwn.net/Articles/249460/) (Mailing list). 6 September 2007. Retrieved 31 March 2015.
89. "Re: Efforts to attract more users?" (http://harmful.cat-v.org/software/c++/rms) (Mailing list). 12 July 2010. Retrieved 31 March 2015.
90. Andrew Binstock (18 May 2011). "Dr. Dobb's: Interview with Ken Thompson" (https://www.drdobbs.com/open-source/interview-with-ken-thompson/229502480). Retrieved 7 February 2014.
91. Peter Seibel (16 September 2009). *Coders at Work: Reflections on the Craft of Programming* (https://books.google.com/books?id=nneBa6-mWfgC&pg=PA475). Apress. pp. 475–476. ISBN 978-1-4302-1948-4.
92. https://gigamonkeys.wordpress.com/2009/10/16/coders-c-plus-plus/
93. https://www.drdobbs.com/architecture-and-design/an-interview-with-donald-knuth/228700500

94. http://tex.loria.fr/litte/knuth-interview
95. Brian Kernighan (18 July 2018). *Brian Kernighan Q&A - Computerphile* (https://www.youtube.com/watch?v=zmYhR8cUX90&t=5m17s).
96. http://www.stroustrup.com/bs_faq.html#really-say-that
97. Pike, Rob (2012). "Less is exponentially more" (https://commandcenter.blogspot.mx/2012/06/less-is-exponentially-more.html).
98. Kreinin, Yossi (13 October 2009). "Defective C++" (https://yosefk.com/c++fqa/defective.html). Retrieved 3 February 2016.

Further reading

- Abrahams, David; Gurtovoy, Aleksey (2005). *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley. ISBN 0-321-22725-5.
- Alexandrescu, Andrei (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley. ISBN 0-201-70431-5.
- Alexandrescu, Andrei; Sutter, Herb (2004). *C++ Design and Coding Standards: Rules and Guidelines for Writing Programs*. Addison-Wesley. ISBN 0-321-11358-6.
- Becker, Pete (2006). *The C++ Standard Library Extensions : A Tutorial and Reference*. Addison-Wesley. ISBN 0-321-41299-0.
- Brokken, Frank (2010). *C++ Annotations* (http://www.icce.rug.nl/documents/cplusplus/). University of Groningen. ISBN 978-90-367-0470-0.
- Coplien, James O. (1994) [reprinted with corrections, original year of publication 1992]. *Advanced C++: Programming Styles and Idioms* (https://archive.org/details/advancedcbsprogr00copl). ISBN 0-201-54855-0.
- Dewhurst, Stephen C. (2005). *C++ Common Knowledge: Essential Intermediate Programming*. Addison-Wesley. ISBN 0-321-32192-8.
- Information Technology Industry Council (15 October 2003). *Programming languages – C++* (Second ed.). Geneva: ISO/IEC. 14882:2003(E).
- Josuttis, Nicolai M. (2012). *The C++ Standard Library, A Tutorial and Reference* (Second ed.). Addison-Wesley. ISBN 978-0-321-62321-8.
- Koenig, Andrew; Moo, Barbara E. (2000). *Accelerated C++ – Practical Programming by Example* (https://archive.org/details/acceleratedcprac2000koen). Addison-Wesley. ISBN 0-201-70353-X.
- Lippman, Stanley B.; Lajoie, Josée; Moo, Barbara E. (2011). *C++ Primer* (https://archive.org/details/cprimer0000lipp_5thed) (Fifth ed.). Addison-Wesley. ISBN 978-0-321-71411-4.
- Lippman, Stanley B. (1996). *Inside the C++ Object Model*. Addison-Wesley. ISBN 0-201-83454-5.
- Meyers, Scott (2005). *Effective C++* (https://archive.org/details/effectivec55spec00meye) (Third ed.). Addison-Wesley. ISBN 0-321-33487-6.
- Stroustrup, Bjarne (2013). *The C++ Programming Language* (Fourth ed.). Addison-Wesley. ISBN 978-0-321-56384-2.
- Stroustrup, Bjarne (1994). *The Design and Evolution of C++*. Addison-Wesley. ISBN 0-201-54330-3.
- Stroustrup, Bjarne (2014). *Programming: Principles and Practice Using C++* (Second ed.). Addison-Wesley. ISBN 978-0-321-99278-9.
- Sutter, Herb (2001). *More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley. ISBN 0-201-70434-X.
- Sutter, Herb (2004). *Exceptional C++ Style*. Addison-Wesley. ISBN 0-201-76042-8.
- Vandevoorde, David; Josuttis, Nicolai M. (2003). *C++ Templates: The complete Guide*. Addison-Wesley. ISBN 0-201-73484-2.

External links

- JTC1/SC22/WG21 (http://www.open-std.org/jtc1/sc22/wg21/) – the ISO/IEC C++ Standard Working Group
- Standard C++ Foundation (https://isocpp.org/) – a non-profit organization that promotes the use and understanding of standard C++. Bjarne Stroustrup is a director of the organization.