Bjarne Stroustrup, Design and Evolution of C++, 1994

4

C++ Language Design Rules

If the map and the terrain disagree,
trust the terrain.
– Swiss army aphorism

Rules for the design of C++ — overall design aims — sociological rules — C++ as a language supporting design — language-technical rules — C++ as a language for low-level programming.

4.1 Rules and Principles

To be genuinely useful and pleasant to work with, a programming language must be designed according to an overall view that guides the design of its individual language features. For C++, this overall view takes the form of a set of rules and constraints. I call them *rules* because I find the term *principles* pretentious in a field as poor in genuine scientific principles as programming language design. Also, to many people the term principle implies the unrealistic implication that no exceptions are acceptable. My rules for the design of C++ most certainly have exceptions. In fact, if a rule and practical experience are in conflict, the rule gives way. This may sound crude, but it is a variant of the principle that theory must account for experimental data or be replaced by a better theory.

These rules cannot be brainlessly applied; nor can they be replaced by a few glib slogans. I saw my job as language designer as deciding which problems needed to be addressed, deciding which problems could be addressed within the framework of C++, and then maintaining balance between the various rules of design for the actual language feature.

The rules guided the working out of features. However, the framework for improvements was provided by the fundamental aims of C++:

Aims:
C++ makes programming more enjoyable for serious programmers.
C++ is a general-purpose programming language that
– is a better C
– supports data abstraction
– supports object-oriented programming

I have organized the rules into four broad sections. The first contains overall ideals for the whole language. These are so general that individual language features don't enter directly into the picture. The second set of rules primarily addresses C++'s role in supporting design. The third addresses technicalities related to the form of the language, and the fourth focuses on C++'s role as a language for low-level systems programming.

The formulation of the rules here has the benefit of hindsight, but the rules and sentiments expressed dominated my thinking from before the completion of the first C++ release in 1985, and – as described in the previous chapters – many of these rules were part of the original conception of C with Classes.

4.2 General Rules

The most general and most important C++ rules have little to do with language-technical issues. They are almost sociological in their focus on the community C++ serves. The nature of the C++ language is largely determined by my choice to serve the current generation of systems programmers solving current problems on current computer systems. Importantly, because the meaning and nature of *current* changes with time, C++ had to evolve to meet the needs of its users; it could not be defined once and for all.

General rules:
C++'s evolution must be driven by real problems.
Don't get involved in a sterile quest for perfection.
C++ must be useful *now*.
Every feature must have a reasonably obvious implementation.
Always provide a transition path.
C++ is a language, not a complete system.
Provide comprehensive support for each supported style.
Don't try to force people.

C++'s evolution must be driven by real problems: In computer science, as in many other fields, we see too many people searching for a problem to apply their pet solution to. I don't know any foolproof way of keeping fads from distorting my view of what is important, but I am acutely aware that many of the language features presented to me as essential are infeasible within the framework of C++ and often irrelevant to real-world programmers.

The right m... ati
to demonstrate how t
fer input from non-re
effort to find and co
avidly looking for so
might help. However
a genuine problem. T
ing a feature.

Don't get involve
is perfect, and none w
ishing a language for
programmers of bene
guage designer of ge
be evolved into irrele
people differ radically
environment is almost
the other hand, progra
code. They need stabi
cal changes are infeasi
Consequently, the nec
feedback and must be
and education. As th
based on tools, techniq

Not every problem
significant enoug
cope directly wit
tor precedence pitfalls
messages.

C++ must be usefu
atively low-powered c
Most programmers ha
have insufficient time t
must be useful to some

Though tempted at
freedom to adjust my
science researchers.

The meaning of thi
as a result of C++'s su
programmers are now
relies on. Further, as p
by programmers change
and more maturity fron
ling (§16) and run-time

Every feature mus

The right motivation for a change to C++ is for several independent programmers to demonstrate how the language is insufficiently expressive for their projects. I prefer input from non-research projects. Whenever possible, I involve real users in the effort to find and complete a solution. I read the programming language literature avidly looking for solutions to such problems and also for general techniques that might help. However, I find the literature wholly unreliable on the subject of what is a genuine problem. Theory itself is never sufficient justification for adding or removing a feature.

Don't get involved in a sterile quest for perfection: No programming language is perfect, and none will ever be as long as problems and systems keep changing. Polishing a language for years trying to reach some notion of perfection simply deprives programmers of benefits from the progress made thus far. It also deprives the language designer of genuine feedback. Without appropriate feedback, a language can be evolved into irrelevance. Problems, computer systems, and – most importantly – people differ radically between environments so that a "perfect fit" to some small environment is almost certainly too specialized to thrive in the larger real world. On the other hand, programmers spend most of their time modifying or interfacing to old code. They need stability to get real work done. Once a language is in real use, radical changes are infeasible, and even small changes are difficult without harming users. Consequently, the necessary quest for significant improvement must rely on genuine feedback and must be accompanied by a serious concern for compatibility, transition, and education. As the language matures, one must increasingly prefer alternatives based on tools, techniques, and libraries over language changes.

Not every problem needs to be solved by C++, and not every problem in C++ is significant enough to warrant a solution. For example, C++ need not be extended to cope directly with pattern matching or theorem proving, and the well-known C operator precedence pitfalls (§2.6.2) are better left alone or addressed through warning messages.

C++ must be useful now: Most programming is relatively mundane, done on relatively low-powered computers, running relatively dated operating systems and tools. Most programmers have less formal training than they would have liked and most have insufficient time to upgrade their knowledge. To serve these programmers, C++ must be useful to someone with average skills, using an average computer.

Though tempted at times, I had no real desire to abandon these people to gain the freedom to adjust my designs to top-of-the-line systems and the tastes of computer science researchers.

The meaning of this rule – like most of the others – changes with time and partly as a result of C++'s success. More powerful computers are now available, and more programmers are now acquainted with the basic concepts and techniques that C++ relies on. Further, as people's ambitions and expectations grow, the problems faced by programmers change. This implies that features requiring more computer resources and more maturity from programmers can and must be considered. Exception handling (§16) and run-time type identification (§14.2) are examples of this.

Every feature must have a reasonably obvious implementation: No feature

should require complicated algorithms for correct or efficient implementation. Ideally, obvious analysis and code-generation strategies should exist, and these should be good enough for real use. If added thought can produce even better results, so much the better. Most features were implemented, used experimentally, and revised before being accepted. Where this pattern was not followed, as in the case of the template instantiation mechanism (§15.10), problems surfaced.

However, there are *many* more users than there are compiler writers, so where there is a real tradeoff between compiler complexity and complexity of use, the resolution must favor the users. I have earned the right to this opinion through years of compiler maintenance.

Always provide a transition path: C++ must grow gradually to serve its users and to benefit from feedback. This implies that great care must be taken to ensure that older code continues to work. When an incompatibility is unavoidable, great care must be taken to help users update their programs. Similarly, there has to be a path from the use of error-prone C-like techniques to a more effective use of C++.

The general strategy for eliminating an unsafe, error-prone, or simply awkward language feature is first to provide a better alternative, then recommend that people avoid the old feature or technique, and only years later – if at all – remove the offending feature. This strategy can be effectively supported by warning messages from the compilers. Often, it is not feasible to eliminate a feature or correct a mistake (the reason is typically the need for C compatibility); the alternative is warnings (§2.6.2). Thus, a C++ implementation can be safer than it appears from the language definition.

C++ is a language, not a complete system: A programming environment has many components. One approach has been to merge all parts into a single, ''integrated'' system. Another approach has been to maintain the classical distinctions between parts of a system such as compilers, linkers, language run-time support libraries, I/O libraries, editors, file systems, databases, etc. C++ follows the latter approach. Through libraries, calling conventions, etc., C++ adapts to the system conventions guiding interoperability of language and tools on each system. This is key for easy portability of implementations and – more importantly – the key to cooperation between code written in different languages. This also allows sharing of tools, eases the cooperation between programmers with different preferences in programming languages, and eases the use of many languages by an individual programmer.

C++ is designed to be one language among many. C++ enables tool development, but does not mandate particular forms. The programmer retains freedom of choice. A key idea is that C++ and its associated tools should ''feel'' right for a given system rather than impose some particular view of what a system and an environment is. This is especially important for large systems and systems with unusual constraints. Such systems are not usually well supported because ''standard'' systems tend to be specialized to serve individuals or small groups doing fairly ''average'' work.

Provide comprehensive support for each supported style: C++ must grow to meet the needs of serious developers. Simplicity is essential, but it is considered relative to the complexity of the projects in which C++ is used. Maintainability and run-time performance of systems written in C++ is considered more important than

keeping the language

It also implies – a must be supported. F data type or object-or classes that take on a use different styles to

Consequently, fea a degree of orthogon is an important sour areas where a more r example, the C++ `virtual` binding, a of techniques relying prefer to see only a fe ''hackery.'' On the applied wherever it d some benefit without

Having a relativel plexity moves from th the language and its b sis, the adoption of ne features must be gradu ting'' or apply all of t make such a gr l know won't hurt

Don't try to force *challenging tasks and* well as from other supp grammers to do ''only grammers will find a v language should suppo than try to force people

This does not impl should try to support e support styles of desig and inheritance. How mum, language mecha added to or subtracted ming.

I am well aware th ple who prefer a more C++ or choose a langu alternatives.

Many programmer

keeping the language definition short. This implies a relatively large language.

It also implies – as experience showed – that many hybrid styles of programming must be supported. People don't just write classes that fit a narrowly defined abstract data type or object-oriented style; they also – often for perfectly good reasons – write classes that take on aspects of both. They also write programs in which different parts use different styles to match needs and taste.

Consequently, features must be designed to be used in combination. This leads to a degree of orthogonality in the design of C++. The opportunity for "unusual" uses is an important source of flexibility and has repeatedly allowed C++ to be used in areas where a more restricted and narrowly focused language would have failed. For example, the C++ rules for access protection, name lookup, `virtual`/non-`virtual` binding, and type are orthogonal. This opens the possibility for a variety of techniques relying on information hiding and derived classes. Some who would prefer to see only a few narrowly defined styles of programming supported deem this "hackery." On the other hand, orthogonality is not a first-order principle; it is applied wherever it doesn't conflict with one of the rules and whenever it provides some benefit without complicating implementations.

Having a relatively large language implies that some of the effort to manage complexity moves from the understanding of libraries and individual programs to learning the language and its basic design techniques. For most people, this change in emphasis, the adoption of new programming techniques, and the application of "advanced" features must be gradual. Few can completely absorb the new techniques "in one sitting" or apply all of their new skills to their work at once (§7.2). C++ is designed to make such a gradual approach feasible and natural. The ideal is: What you don't know won't hurt you. The static type system and compiler warning messages help.

Don't try to force people: Programmers are smart people. They are engaged in challenging tasks and need all the help they can get from a programming language as well as from other supporting tools and techniques. Trying to seriously constrain programmers to do "only what is right" is inherently wrongheaded and will fail. Programmers will find a way around rules and restrictions they find unacceptable. The language should support a range of reasonable design and programming styles rather than try to force people into adopting a single notion.

This does not imply that all ways of programming are equally good or that C++ should try to support every kind of programming style. C++ was designed to directly support styles of design relying on extensive static type checking, data abstraction, and inheritance. However, moralizing over how to use the features is kept to a minimum, language mechanisms are as far as possible kept policy free, and no feature is added to or subtracted from C++ exclusively to prevent a coherent style of programming.

I am well aware that not everyone appreciates choice and variety. However, people who prefer a more restrictive environment can impose one through style rules in C++ or choose a language designed to provide the programmer with a smaller set of alternatives.

Many programmers particularly dislike being told that something might be an

error when it happens not to be. Consequently, "potential errors" are not errors in C++. For example, it is not an error to write declarations that will allow an ambiguous use. The error is an ambiguous use, not the mere possibility of such an error. In my experience, most "potential errors" never manifest themselves so to defer the error message is to avoid giving it. Much convenience and flexibility result from such deferrals.

4.3 Design Support Rules

The rules listed here relate primarily to C++'s role in supporting design based on notions of data abstraction and object-oriented programming. That is, they are more concerned with the language's role as a support for thinking and expression of high-level ideas than its role as a "high-level assembler" along the lines of C or Pascal.

Design support rules:
Support sound design notions.
Provide facilities for program organization.
Say what you mean.
All features must be affordable.
It is more important to allow a useful feature than to prevent every misuse.
Support composition of software from separately developed parts.

Support sound design notions: Each individual language feature must fit into an overall pattern. That overall pattern must help answer questions of what abilities are desirable. The language itself cannot provide that; the guiding pattern must come from a different conceptual level. For C++, that level is provided by ideas of how programs can be designed.

My aim is to raise the level of abstraction in systems programming in a way similar to what C did by replacing assembler as the mainstay of systems work. Ideas for new features are considered in light of how they might enhance C++ as a language for expressing designs. In particular, individual features are considered in light of how they can make the notion that a concept is represented by a class effective. This is the key to C++'s support for data abstraction and object-oriented programming.

A programming language is not and should not be a complete design language. A design language should be richer and less concerned with details than a language suitable for systems programming must be. However, the programming language should support some notions of design as directly as possible to ease communication between designers and programmers (who are often the same people "wearing different hats") and to simplify tool building.

Viewing the programming language in terms of design techniques allows suggested language features to be accepted or excluded based on their relationship to the design styles supported. No language can support every style, and a language supporting only one narrowly defined design philosophy will fail for lack of adaptability. Enhancing C++ to support the continuum of design techniques that map into the

"better C" / data ab
the temptation to try t
stimulus to improvem

Provide facilities
nize programs to be e
problem solved by C.
sion and statement p
elsewhere. Whenever
has been evaluated ba
made the expression c
allowing declarations
expressions and staten

Say what you me
gap between what pec
express directly in the
appears in a mess of b

The primary mean
declarative. Almost e
declarative and then e
of silly errors, and imp

Where a declarati
often help. The alloc
(§14.3) are examples.
sion of intent was "to
rather than in the
guage in general, and i
ble than earlier general

All features must
guage feature or reco
must also be affordabl
tive jet," may be a val
but to all but millionair

A feature was add
functionality at signifi
given the choice of do
ciency unless there is
were provided to allow
behaved alternative to
elegant and efficient. \
if it is deemed essential

It is more importa
You can write bad pro
of accidental misuse of
the default behavior o

"better C" / data abstraction / object-oriented programming spectrum helped avoid the temptation to try to make C++ everything to all people while providing a constant stimulus to improvements.

Provide facilities for program organization: Compared to C, C++ helps organize programs to be easier to write, read, and maintain. I considered computation a problem solved by C. Like just about everybody else, I have ideas of how the expression and statement part of C could be improved, but I decided to focus my efforts elsewhere. Whenever a new kind of expression or statement has been suggested, it has been evaluated based on whether it affected the structure of the program or merely made the expression of some local computation easier. With few exceptions, such as allowing declarations to appear where a variable is first needed (§3.11.5), the C expressions and statements have been left unchanged.

Say what you mean: The fundamental problem with lower-level languages is the gap between what people can express when they talk to each other and what they can express directly in the programming language. The basic structure of a program disappears in a mess of bits, bytes, pointers, loops, etc.

The primary means of narrowing this semantic gap is to make a language more declarative. Almost every facility provided by C++ hinges on making something declarative and then exploiting the added structure in consistency checking, detection of silly errors, and improved code generation.

Where a declarative structure cannot be employed, a more explicit notation can often help. The allocation/deallocation operators (§10.2) and the new cast syntax (§14.3) are examples. An early expression of the ideal of direct and explicit expression of intent was "to allow expression of all important things in the language itself rather than in the comments or through macro hackery." This implies that the language in general, and its type system in particular, must be more expressive and flexible than earlier general-purpose languages.

All features must be affordable: It is not enough to provide a user with a language feature or recommend a technique for some problem. The solution offered must also be affordable. Otherwise, the advice is almost an insult: "Rent an executive jet," may be a valid response to, "What is the best way of getting to Memphis?" but to all but millionaires, it is not a very helpful answer.

A feature was added to C++ only when there was no way of achieving similar functionality at significantly lesser cost. My experience is that if programmers are given the choice of doing something efficiently or elegantly, most will choose efficiency unless there is an obvious major reason not to. For example, inline functions were provided to allow cost-free crossing of protection boundaries and to be a better-behaved alternative to many uses of macros. The ideal is of course for facilities to be elegant and efficient. Where that is not feasible, the facility either isn't provided or – if it is deemed essential – it is provided efficiently.

It is more important to allow a useful feature than to prevent every misuse: You can write bad programs in *any* language. It is important to minimize the chance of accidental misuse of features, and much effort has been spent trying to ensure that the default behavior of C++ constructs is either sensible or leads to compile-time

errors. For example, by default all function argument types are checked – even across separate compilation boundaries – and by default, all class members are private. However, a systems programming language cannot prevent a determined programmer from breaking the system so design effort is better expended providing facilities for writing good programs than preventing the inevitable bad ones. In the longer run, programmers seem to learn. This is a variant of the old C ''trust the programmer'' slogan. The various type checking and access control rules exist to allow a class provider to state clearly what is expected from users, to protect against accidents. Those rules are not intended as protection against deliberate violation (§2.10).

Support composition of software from separately developed parts: Programmers need more support for complex applications than simple ones, more support for large programs than small ones, and more support for applications under efficiency constraints than applications with ample resources. Much of the effort in the design of C++ was spent addressing the first two of those observations under the constraints of the third. As applications get larger and more complex, they must be composed out of semi-independent parts to be manageable.

Anything that allows a component of a larger system to be developed independently and then used without modification in a larger system serves this purpose. Much of the evolution of C++ has been driven by that idea. Classes themselves are the original such C++ feature, and abstract classes (§13.2.2) explicitly support separation between interfaces and implementations. In fact, classes can be used to express a continuum of coupling strategies [Stroustrup,1990b]. Exceptions allow error handling to be decoupled from a library (§16.1), templates allow composition based on types (§15.3, §15.6, §15.8), namespaces solve the namespace pollution problem (§17.2), and run-time type identification addresses the problem of what to do when the exact type of an object has been ''lost'' by passing it through a library (§14.2.1).

The notion that programmers need more support when developing larger systems implies that efficiency mustn't be compromised by reliance on optimization techniques that work best for small programs. Consequently, object layout can be determined given a single compilation unit in isolation, and virtual function calls can be compiled into efficient code without relying on cross-compilation-unit optimizations. This is true even when *efficient* means efficiently compared to C. Further optimizations are possible when information about a complete program is available. For example, looking at a complete program and a call of a virtual function, one can – in the absence of dynamic linking – sometimes determine the actual function called. In that case, one can call replace the virtual function call with an ordinary function call or even inline. C++ implementations that can do that exist. However, such optimizations are not necessary for generating efficient code; they are simply an added benefit when run-time efficiency is preferred to compile-time efficiency and dynamic linking of new derived classes. When such global optimization is not deemed reasonable, a virtual function call can still be optimized away when the virtual function is applied to an object of known type; even Cfront Release 1.0 did that.

Support for larger systems is often discussed under the heading ''support for libraries'' (§8).

4.4 Language-l

The following rules
questions of what ca

No implicit v
Provide as go
Locality is go
Avoid order d
If in doubt, pi
Syntax matter
Preprocessor

No implicit viola
specific type such as
way that is inconsiste
guage where such vi
every such violation i

C++ inherits featu
sible to detect every v
violation of the type
an explicitly unchecke
system. Any use of t
ing. More importantl
venient and equ
are derived class
and dynamically chec
common practice, the
grammers have yet to

Wherever possible
that cannot be checke
checked at link time.
are provided to help t
linker cannot catch.
dependable, though.

Provide as good
user-defined types are
port as possible from t
be allocated only on th
local variables for ar
oriented types (concret

Locality is good:
contained except whe
such services to be a

4.4 Language-Technical Rules

The following rules address questions of how things are expressed in C++ rather than questions of what can be expressed.

Language-technical rules:
No implicit violations of the static type system.
Provide as good support for user-defined types as for built-in types.
Locality is good.
Avoid order dependencies.
If in doubt, pick the variant of a feature that is easiest to teach.
Syntax matters (often in perverse ways).
Preprocessor usage should be eliminated.

No implicit violations of the static type system: Every object is created with a specific type such as `double`, `char*`, or `dial_buffer`. If an object is used in a way that is inconsistent with its given type the type system has been violated. A language where such violation can never happen is strongly typed. A language where every such violation is detected at compile time is strongly statically typed.

C++ inherits features from C, such as unions, casts, and arrays, that make it impossible to detect every violation at compile time. Currently, C++ does not admit implicit violation of the type system. That is, you need to explicitly use a union, cast, array, an explicitly unchecked function argument, or explicitly unsafe C linkage to break the system. Any use of the unsafe features can be made to cause a (compile time) warning. More importantly, C++ now possesses language features that make it more convenient and equally efficient to avoid the unsafe features than to use them. Examples are derived classes (§2.9), a standard array template (§8.5), type-safe linkage (§11.3), and dynamically checked casts (§14.2). Because of C compatibility requirements and common practice, the path to this state of affairs has been long and hard; most programmers have yet to adopt the safer practices.

Wherever possible, checking is done at compile time. Wherever possible, things that cannot be checked given only the information in a single compilation units are checked at link time. Finally, run-time type information (§14.2) and exceptions (§16) are provided to help the programmer cope with error conditions that a compiler and a linker cannot catch. Where applicable, compile-time checking is cheaper and more dependable, though.

Provide as good support for user-defined types as for built-in types: Since user-defined types are intended to be central to C++ programs, they need as much support as possible from the language. Therefore, restrictions such as "class objects can be allocated only on the free store" were not acceptable. The need to provide genuine local variables for arithmetic types such as `complex` led to support for value-oriented types (concrete types) comparable to or even superior to the built-in types.

Locality is good: When writing a piece of code, one would prefer it to be self-contained except where it needs a service from elsewhere. One would also prefer such services to be available without too much fuss and bother. Conversely, one

would like to supply functions, classes, etc., to others without fear of interference between implementation details and other people's code.

C is about as far from these ideals as one can get. Every global function and variable name is visible to the linker and will clash with other uses of the same name unless explicitly declared `static`. Every name can be used as a function name without previous declaration. As a relic of the days when the names of structure members were global, the names of structures declared within structures are global. In addition, the preprocessor's macro processing doesn't respect scope, so any sequence of characters in the program text just might be changed into something different if a change is made to a header file or a compiler option (§18.1). All this adds up to very powerful stuff if you want to affect the meaning of some apparently local code or want to affect the rest of the world by a small "local" change. On average, I consider this most disruptive to my comprehension of complex software and to maintenance. Consequently, I set out to provide better insulation against disruptions from "elsewhere" and better control over what is "exported" from my code.

Classes provide the first and most important mechanisms for localizing code and channeling access through a well-defined interface. Nested classes (§3.12, §13.5) and namespaces (§17) extend notions of local scope and explicit granting of access further. In each case, the amount of global information in a system decreases significantly.

Access control localizes access without imposing run-time or space overheads needed for complete decoupling (§2.10). Abstract classes allow a greater degree of decoupling at minimal cost (§13.2).

Within classes and namespaces, it is important that people can separate the declarations from the implementations, thus making it easier to see what a class does without having to skip past function bodies specifying how it is done. Inline functions in class declarations are allowed so that locality can be achieved when this separation is not helpful.

Finally, code is easier to understand and manipulate if significant chunks fit on a screen. C's traditional terseness helps here, and the C++ rules that allow new variables to be introduced where they are first needed (§3.11.5) is a further step in this direction.

Avoid order dependencies: An order dependence is an opportunity for confusion and for errors when code is reorganized. People are aware that statements are executed in a definite order, but dependencies between global declarations and between class member declarations are often overlooked. The overloading rules (§11.2) and the rules for the use of base classes (§12.2) were specifically crafted to avoid order dependencies. Ideally, it should be an error if the reversal of the order of two declarations could cause a different meaning. That is the rule for class members (§6.3.1), but it cannot be imposed for global declarations. The C preprocessor can wreak havoc by introducing unexpected and ill-behaved dependencies through macro processing (§18.1).

I sometime express my desire to avoid subtle resolutions by saying, "It is not the compiler's job to make up your mind for you." In other words, a compile-time error

is more acceptable t
inheritance are a goo
functions are an exam
bility and flexibility (

If in doubt, pick
ondary rule for choos
an argument for logic
and reference manual
a practical applicatio
support personnel. It
plicity mustn't be ach

Syntax matters (
coherent and in gener
Syntax is a secondary
lutely any syntax.

However, syntax i
face. People are dev
curious fanaticism. I
notions and design id
Consequently, the C+
prejudices, while aimi
aim is to fade out wa
while minimizing the
(§2.8.1).

My experien t
the point where a cor
teach. This effect is n
dislike for new keyw
choose the new keywo

I try to make sign
problem with old-styl
make semantically ug
match (§14.3.3). In g

Preprocessor usa
and later C++ would
ciently expressive and
the other hand, the ug
advanced and elegan
expensive to build and

Consequently, alte
found for every esse
improved C++ progra
of many difficult bugs
namespaces (§17) are

is more acceptable than an obscure resolution. The ambiguity rules for multiple inheritance are a good example of this (§12.2). The ambiguity rules for overloaded functions are an example of how hard this is to achieve under constraints of compatibility and flexibility (§11.2.2).

If in doubt, pick the variant of a feature that is easiest to teach: This is a secondary rule for choosing between alternatives. It is tricky to apply because it can be an argument for logical beauty and also for sticking to the familiar. Writing tutorials and reference manual descriptions to see how easy they are for people to understand is a practical application of this rule. One intent is to ease the task for educators and support personnel. It is important to remember that programmers are not stupid; simplicity mustn't be achieved at the expense of important functionality.

Syntax matters (often in perverse ways): It is essential to have the type system coherent and in general to have the semantics of the language clean and well defined. Syntax is a secondary issue, and it appears that programmers can learn to love absolutely any syntax.

However, syntax is what people see. Syntax is the language's primary user interface. People are devoted to certain forms of syntax and express their opinions with curious fanaticism. I see no hope of changing this or introducing new semantic notions and design ideas in the face of emotional opposition to a particular syntax. Consequently, the C++ syntax is crafted with care to avoid offending programmers' prejudices, while aiming to make the syntax more rational and regular over time. My aim is to fade out warts such as implicit `int` (§2.8.1) and old-style casts (§14.3.1), while minimizing the use of the more complicated forms of the declarator syntax (§2.8.1).

My experience is that people are addicted to keywords for introducing concepts to the point where a concept that doesn't have its own keyword is surprisingly hard to teach. This effect is more important and deep-rooted than people's vocally expressed dislike for new keywords. Given a choice and time to consider, people invariably choose the new keyword over a clever workaround.

I try to make significant operations highly visible. For example, one significant problem with old-style casts is that they are almost invisible. In addition, I prefer to make semantically ugly operations, such as ill-behaved casts, syntactically ugly to match (§14.3.3). In general, verbosity is avoided.

Preprocessor usage should be eliminated: Without the C preprocessor, C itself and later C++ would have been stillborn. Without Cpp, they simply weren't sufficiently expressive and flexible to handle every task needed in significant projects. On the other hand, the ugly and low-level semantics of Cpp are the primary reason more advanced and elegant C programming environments have been too difficult and expensive to build and use.

Consequently, alternatives that fit with the syntax and semantics of C++ must be found for every essential Cpp feature. That done, we'll get cheaper and much improved C++ programming environments. Along the way, we'll root out the sources of many difficult bugs. Templates (§15), inline functions (§2.4.1), `const` (§3.8), and namespaces (§17) are steps on the way.

4.5 Low-Level Programming Support Rules

Naturally, the rules mentioned above apply to essentially all language features. The rules below also affect C++ as a language for expressing high-level designs.

Low-level programming support rules:
Use traditional (dumb) linkers.
No gratuitous incompatibilities with C.
Leave no room for a lower-level language below C++ (except assembler).
What you don't use, you don't pay for (zero-overhead rule).
If in doubt, provide means for manual control.

Use traditional (dumb) linkers: Ease of porting and ease of cooperation with software written in other languages were early goals. Insisting that C++ should be implementable with traditional linkers ensures that. Having to manage with linker technology that dates from early Fortran days can be painful, though. Several features of C++, notably type-safe linkage (§11.3) and templates (§15), can be implemented using traditional linkers, but they can be implemented better with more linker support. A secondary aim has been for C++ to provide a stimulus to improved linker design.

Using traditional linkers makes it relatively easy to maintain link compatibility with C. This is essential for smooth use of operating system facilities, for using C, Fortran, etc., libraries, and for writing code to be used as libraries from other languages. Using traditional linkers is also essential for writing code intended to be part of the lower levels of a system, such as device drivers.

No gratuitous incompatibilities with C: C is the most successful systems programming language ever. Hundreds of thousands of programmers know C well, billions of lines of C exist, and a tools and services industry focused on C exists. C++ is based on C. The question is, "How closely should the C++ definition match that of C?" C++ doesn't aim at 100% compatibility with C because that would have compromised the aims of type safety and support for design. However, where these aims are not interfered with incompatibilities are avoided – even at the cost of inelegance. In most cases, C incompatibilities have been accepted only when a C rule left a gaping hole in the type system.

Over the years, C++'s greatest strength and its greatest weakness has been its C compatibility. This came as no surprise. The degree of C compatibility will be a major issue in the future. Over the coming years, C compatibility will become less and less of an advantage and more and more of a liability. A path of evolution must be provided (§9).

Leave no room for a lower-level language below C++ (except assembler): If a language aims at being truly high level – that is, it completely protects its programmers from the ugly and boring details of the underlying computer – it must relinquish the dirtier tasks of systems programming to some other language. Typically, that language has been C. Typically, C has then replaced the higher-level language in most areas where control or speed were deemed essential. Often, this has led to a system programmed completely in C or to one that could only be mastered by someone who

knows both language
difficult choice of wl
has to keep the prim
providing low-level f
systems out of both.

To remain a viabl
to access hardware c
operation and data ty
native is to use C or
features and render th
details. The aim is tc
ing undue burdens.

What you don't
have a well-earned re
that the overhead of
the features in the lar
needed for various k
because some features
elaborated to accomn
tributed fat" was dee
lower-level language l
and high-performance

This rule has repea
(§3.5), multiple inherit
tion handling, ar n
In each case, the ...ati
mentation that obeyed
implementer can decid
other desirable propert
programmers react har

Of all the rules, th
edge when it comes to

If in doubt, pro
"advanced technology
sophisticated will be u
example of this (§2.4.
been more careful and
detailed control of men
made through manual
expense of getting in th

knows both languages well. In the latter case, a programmer is too often left with a difficult choice of which level of programming is most suitable for a given task and has to keep the primitives and principle of both in mind. C++ tried another path by providing low-level features, abstraction mechanisms, and support for creating hybrid systems out of both.

To remain a viable systems programming language, C++ must maintain C's ability to access hardware directly, to control data structure layout, and to have primitive operation and data types that map on to hardware in a one-to-one fashion. The alternative is to use C or assembler. The language design task is to isolate the low-level features and render them unnecessary for code that doesn't deal directly with system details. The aim is to protect programmers against accidental misuse without imposing undue burdens.

What you don't use, you don't pay for (zero-overhead rule): Large languages have a well-earned reputation for generating large and slow code. The usual reason is that the overhead of supporting supposedly advanced features is distributed over all the features in the language. For example, all objects are large to hold information needed for various kinds of housekeeping, indirect access is imposed on all data because some features are best managed through indirections, or control structures are elaborated to accommodate "advanced control abstractions." This kind of "distributed fat" was deemed unsuitable for C++. Accepting it would leave room for a lower-level language below C++ and make C a better choice than C++ for low-level and high-performance work.

This rule has repeatedly been crucial for C++ design decisions. Virtual functions (§3.5), multiple inheritance (§12.4.2), run-time type identification (§14.2.2.2), exception handling, and templates are all features that owe part of their design to this rule. In each case, the feature was accepted only after I convinced myself that an implementation that obeyed the zero-overhead rule could be constructed. Naturally, an implementer can decide to make a tradeoff between the zero-overhead rule and some other desirable property of a system, but this has to be done very carefully. Many programmers react harshly and emotionally to distributed fat.

Of all the rules, the zero-overhead rule is probably the one that has the sharpest edge when it comes to rejecting a suggested feature.

If in doubt, provide means for manual control: I am reluctant to trust "advanced technology" and particularly loath to assume that something really sophisticated will be universally and cheaply available. Inline functions are a good example of this (§2.4.1). Template instantiation is an example where I should have been more careful and later had to add a mechanism for explicit control (§15.10). The detailed control of memory management is an example of where important gains were made through manual control, yet only time will tell if these gains were made at the expense of getting in the way of automated techniques (§10.7).

4.6 A Final Word

All of these rules must be taken into account for a major language feature. Leaving one out would most likely lead to an imbalance that could hurt a group of users. Similarly, letting one rule dominate at the expense of others would cause similar problems.

I have tried to keep my rules positive and prescriptive rather than building up a list of prohibitions. This makes it inherently more difficult to exclude new ideas. My view of C++ as a language for production software and a focus on facilities that affect program structure counteracts the natural tendency to make minor adjustments.

A more specific and detailed list of issues considered for a language feature is the checklist suggested by the ANSI/ISO committee's working group for extensions (§6.4.1).

Post-Release-1.
The Annotated
ARM feature o
ture overview.

5.1 Introductio

Part II presents featu
language features rat

The reason to de
order was not import
the language was goi
tures might be neede
down and do it all in
long and would have
quently, extensions w
order was of crucial
language coherent at
shape of C++. Pres
obscure the logical st