Evolving a language in and for the real world: C++ 1991-2006

Bjarne Stroustrup
Texas A&M University
www.research.att.com/~bs

Abstract

This paper outlines the history of the C++ programming language from the early days of its ISO standardization (1991), through the 1998 ISO standard, to the later stages of the C++0x revision of that standard (2006). The emphasis is on the ideals, constraints, programming techniques, and people that shaped the language, rather than the minutiae of language features. Among the major themes are the emergence of generic programming and the STL (the C++ standard library's algorithms and containers). Specific topics include separate compilation of templates, exception handling, and support for embedded systems programming. During most of the period covered here, C++ was a mature language with millions of users. Consequently, this paper discusses various uses of C++ and the technical and commercial pressures that provided the background for its continuing evolution.

Categories and Subject Descriptors K.2 [*History of Computing*]: systems

General Terms Design, Programming Language, History

Keywords C++, language use, evolution, libraries, standardization, ISO, STL, multi-paradigm programming

1. Introduction

In October 1991, the estimated number of C++ users was 400,000 [121]. The corresponding number in October 2004 was 3,270,000 [61]. Somewhere in the early '90s C++ left its initial decade of exponential growth and settled into a decade of steady growth. The key efforts over that time were to

1. use the language (obviously)
2. provide better compilers, tools, and libraries
3. keep the language from fragmenting into dialects
4. keep the language and its community from stagnating

DOI 10.1145/1238844.1238848
http://doi.acm.org/10.1145/1238844.1238848

Obviously, the C++ community spent the most time and money on the first of those items. The ISO C++ standards committee tends to focus on the second with some concern for the third. My main effort was on the third and the fourth. The ISO committee is the focus for people who aim to improve the C++ language and standard library. Through such change they (we) hope to improve the state of the art in real-world C++ programming. The work of the C++ standards committee is the primary focus of the evolution of C++ and of this paper.

Thinking of C++ as a platform for applications, many have wondered why — after its initial success — C++ didn't shed its C heritage to "move up the food chain" and become a "truly object-oriented" applications programming language with a "complete" standard library. Any tendencies in that direction were squelched by a dedication to systems programming, C compatibility, and compatibility with early versions of C++ in most of the community associated with the standards committee. Also, there were never sufficient resources for massive projects in that community. Another important factor was the vigorous commercial opportunism by the major software and hardware vendors who each saw support for application building as *their* opportunity to distinguish themselves from their competition and to lock in their users. Minor players saw things in much the same light as they aimed to grow and prosper. However, there was no lack of support for the committee's activities within its chosen domain. Vendors rely on C++ for their systems and for highly demanding applications. Therefore, they have provided steady and most valuable support for the standards effort in the form of hosting and — more importantly — of key technical people attending.

For good and bad, ISO C++ [66] remains a general-purpose programming language with a bias towards systems programming that

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming

An explanation of the first three items from a historical perspective can be found in [120, 121]; the explanation of "supports generic programming" is a major theme of this paper. Bringing aspects of generic programming into the mainstream is most likely C++'s greatest contribution to the software development community during this period.

The paper is organized in loose chronological order:

- §**1** Introduction
- §**2** Background: C++ 1979-1991 — early history, design criteria, language features.
- §**3** The C++ world in 1991 — the C++ standards process, chronology.
- §**4** Standard library facilities 1991-1998 — the C++ standard library with a heavy emphasis on its most important and innovative component: the STL (containers, algorithms, and iterators).
- §**5** Language features 1991-1998 — focusing on separate compilation of templates, exception safety, run-time type information and namespaces.
- §**6** Standards maintenance 1997-2003 — for stability, no additions were made to the C++ standard. However, the committee wasn't idle.
- §**7** C++ in real-world use — application areas; applications programming vs. systems programming; programming styles; libraries, Application Binary Interfaces (ABIs), and environments; tools and research; Java, C#, and C; dialects.
- §**8** C++0x — aims, constraints, language features, and library facilities.
- §**9** Retrospective — influences and impact; beyond C++.

The emphasis is on the early and later years: The early years shaped current C++ (C++98). The later ones reflect the response to experience with C++98 as represented by C++0x. It is impossible to discuss how to express ideas in code without code examples. Consequently, code examples are used to illustrate key ideas, techniques, and language facilities. The examples are explained to make them accessible to non-C++ programmers. However, the focus of the presentation is the people, ideas, ideals, techniques, and constraints that shape C++ rather than on language-technical details. For a description of what ISO C++ is today, see [66, 126]. The emphasis is on straightforward questions: What happened? When did it happen? Who were there? What were their reasons? What were the implications of what was done (or not done)?

C++ is a living language. The main aim of this paper is to describe its evolution. However, it is also a language with a great emphasis on backwards compatibility. The code that I describe in this paper still compiles and runs today. Consequently, I tend to use the present tense to describe it. Given the emphasis on compatibility, the code will probably also run as described 15 years from now. Thus, my use of the present tense emphasizes an important point about the evolution of C++.

2. Background: C++ 1979-1991

The early history of C++ (up until 1991) is covered by my HOPL-II paper [120]. The standard reference for the first 15 years of C++ is my book *The Design and Evolution of C++*, usually referred to as D&E [121]. It tells the story from the pre-history of C++ until 1994. However, to set the scene for the next years of C++, here is a brief summary of C++'s early history.

C++ was designed to provide Simula's facilities for program organization together with C's efficiency and flexibility for systems programming. It was intended to deliver that to real projects within half a year of the idea. It succeeded.

At the time, I realized neither the modesty nor the preposterousness of that goal. The goal was modest in that it did not involve innovation, and preposterous in both its time scale and its Draconian demands on efficiency and flexibility. While a modest amount of innovation did emerge over the early years, efficiency and flexibility have been maintained without compromise. While the goals for C++ have been refined, elaborated, and made more explicit over the years, C++ as used today directly reflects its original aims.

Starting in 1979, while in the Computer Science Research Center of Bell Labs, I first designed a dialect of C called "C with Classes". The work and experience with C with Classes from 1979 to 1983 determined the shape of C++. In turn, C with Classes was based on my experiences using BCPL and Simula as part of my PhD studies (in distributed systems) in the University of Cambridge, England. For challenging systems programming tasks I felt the need for a "tool" with the following properties:

- A good tool would have Simula's support for program organization – that is, classes, some form of class hierarchies, some form of support for concurrency, and compile-time checking of a type system based on classes. This I saw as support for the process of inventing programs; that is, as support for design rather than just support for implementation.
- A good tool would produce programs that ran as fast as BCPL programs and share BCPL's ability to combine separately compiled units into a program. A simple linkage convention is essential for combining units written in languages such as C, Algol68, Fortran, BCPL, assembler, etc., into a single program and thus not to suffer the handicap of inherent limitations in a single language.
- A good tool should allow highly portable implementations. My experience was that the "good" implementation I needed would typically not be available until "next year" and only on a machine I couldn't afford. This implied that a tool must have multiple sources of implementations (no monopoly would be sufficiently responsive to users of "unusual" machines and to poor graduate students), that there should be no complicated run-time support system to port, and that there should be only very

limited integration between the tool and its host operating system.

During my early years at Bell Labs, these ideals grew into a set of "rules of thumb" for the design of C++.

- 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 to use a specific programming style.
- 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.
- 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.
- 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.

These criteria are explored in detail in Chapter 4 of D&E [121]. C++ as defined at the time of release 2.0 in 1989 strictly fulfilled these criteria; the fundamental tensions in the effort to design templates and exception-handling mechanisms for C++ arose from the need to depart from some aspects of these criteria. I think the most important property of these criteria is that they are only loosely connected with specific programming language features. Rather, they specify constraints on solutions to design problems.

Reviewing this list in 2006, I'm struck by two design criteria (ideals) that are not explicitly stated:

- There is a direct mapping of C++ language constructs to hardware
- The standard library is specified and implemented in C++

Coming from a C background and being deep in the development of the C++98 standard (§3.1), these points (at the time, and earlier) seemed so obvious that I often failed to emphasize them. Other languages, such as Lisp, Smalltalk, Python, Ruby, Java, and C#, do not share these ideals. Most languages that provide abstraction mechanisms still have to provide the most useful data structures, such as strings, lists, trees, associative arrays, vectors, matrices, and sets, as built-in facilities, relying on other languages (such as assembler, C, and C++) for their implementation. Of major languages, only C++ provides general, flexible, extensible, and efficient containers implemented in the language itself. To a large extent, these ideals came from C. C has those two properties, but not the abstraction mechanisms needed to define nontrivial new types.

The C++ ideal – from day one of C with Classes (§2.1) – was that the language should allow optimal implementation of arbitrary data structures and operations on them. This constrained the design of abstraction mechanisms in many useful ways [121] and led to new and interesting implementation techniques (for example, see §4.1). In turn, this ideal would have been unattainable without the "direct mapping of C++ language constructs to hardware" criterion. The key idea (from C) was that the operators directly reflected hardware operations (arithmetic and logical) and that access to memory was what the hardware directly offered (pointers and arrays). The combination of these two ideals is also what makes C++ effective for embedded systems programming [131] (§6.1).

2.1 The Birth of C with Classes

The work on what eventually became C++ started with an attempt to analyze the UNIX kernel to determine to what extent it could be distributed over a network of computers connected by a local area network. This work started in April of 1979 in the Computing Science Research Center of Bell Laboratories in Murray Hill, New Jersey. Two subproblems soon emerged: how to analyze the network traffic that would result from the kernel distribution and how to modularize the kernel. Both required a way to express the module structure of a complex system and the communication pattern of the modules. This was exactly the kind of problem that I had become determined never to attack again without proper tools. Consequently, I set about developing a proper tool according to the criteria I had formed in Cambridge.

In October of 1979, I had the initial version of a pre-processor, called Cpre, that added Simula-like classes to C. By March of 1980, this pre-processor had been refined to the point where it supported one real project and several experiments. My records show the pre-processor in use on 16 systems by then. The first key C++ library, called "the task system", supported a coroutine style of programming [108, 115]. It was crucial to the usefulness of "C with Classes," as the language accepted by the pre-processor was called, in most early projects.

During the April to October period the transition from thinking about a "tool" to thinking about a "language" had occurred, but C with Classes was still thought of primarily as an extension to C for expressing modularity and concurrency. A crucial decision had been made, though. Even though support of concurrency and Simula-style simulations was a primary aim of C with Classes, the language contained no primitives for expressing concurrency; rather, a combination of inheritance (class hierarchies) and the ability to define class member functions with special meanings recognized by the pre-processor was used to write the library that supported the desired styles of concurrency. Please note that "styles" is plural. I considered it crucial — as I still do — that more than one notion of concurrency should be expressible in the language.

Thus, the language provided general mechanisms for organizing programs rather than support for specific application areas. This was what made C with Classes and later C++ a general-purpose language rather than a C variant with extensions to support specialized applications. Later, the choice between providing support for specialized applications or general abstraction mechanisms has come up repeatedly. Each time, the decision has been to improve the abstraction mechanisms.

2.2 Feature Overview

The earliest features included classes, derived classes, public/private access control, type checking and implicit conversion of function arguments. In 1981, inline functions, default arguments, and the overloading of the assignment operator were added based on perceived need.

Since a pre-processor was used for the implementation of C with Classes, only new features, that is features not present in C, needed to be described and the full power of C was directly available to users. Both of these aspects were appreciated at the time. In particular, having C as a subset dramatically reduced the support and documentation work needed. C with Classes was still seen as a dialect of C. Furthermore, classes were referred to as "An Abstract Data Type Facility for the C Language" [108]. Support for object-oriented programming was not claimed until the provision of virtual functions in C++ in 1983 [110].

A common question about "C with Classes" and later about C++ was "Why use C? Why didn't you build on, say, Pascal?" One version of my answer can be found in [114]:

> C is clearly not the cleanest language ever designed nor the easiest to use, so why do so many people use it?
>
> - C is *flexible*: It is possible to apply C to most every application area, and to use most every programming technique with C. The language has no inherent limitations that preclude particular kinds of programs from being written.
> - C is *efficient*: The semantics of C are 'low level'; that is, the fundamental concepts of C mirror the fundamental concepts of a traditional computer. Consequently, it is relatively easy for a compiler and/or a programmer to efficiently utilize hardware resources for a C program.
> - C is *available*: Given a computer, whether the tiniest micro or the largest super-computer, the chance is that there is an acceptable-quality C compiler available and that that C compiler supports an acceptably complete and standard C language and library. There are also libraries and support tools available, so that a programmer rarely needs to design a new system from scratch.
> - C is *portable*: A C program is not automatically portable from one machine (and operating system) to another nor is such a port necessarily easy to do. It is, however, usually possible and the level of difficulty is such that porting even major pieces of software with inherent machine dependences is typically technically and economically feasible.
>
> Compared with these 'first-order' advantages, the 'second-order' drawbacks like the curious C declarator syntax and the lack of safety of some language constructs become less important.

Pascal was considered a toy language [78], so it seemed easier and safer to add type checking to C than to add the features considered necessary for systems programming to Pascal. At the time, I had a positive dread of making mistakes of the sort where the designer, out of misguided paternalism or plain ignorance, makes the language unusable for real work in important areas. The ten years that followed clearly showed that choosing C as a base left me in the mainstream of systems programming where I intended to be. The cost in language complexity has been considerable, but (just barely) manageable. The problem of maintaining compatibility with an evolving C language and standard library is a serious one to this day (see §7.6).

In addition to C and Simula, I considered Modula-2, Ada, Smalltalk, Mesa, and Clu as sources for ideas for C++ [111], so there was no shortage of inspiration.

2.3 Work Environment

C with Classes was designed and implemented by me as a research project in the Computing Science Research Center of Bell Labs. This center provided a possibly unique environment for such work. When I joined in 1979, I was basically told to "do something interesting," given suitable computer resources, encouraged to talk to interesting and competent people, and given a year before having to formally present my work for evaluation.

There was a cultural bias against "grand projects" requiring many people, against "grand plans" like untested paper designs for others to implement, and against a class distinction between designers and implementers. If you liked such things, Bell Labs and other organizations had many places where you could indulge such preferences. However, in the Computing Science Research Center it was almost a requirement that you — if you were not into theory — personally implemented something embodying your ideas and found users who could benefit from what you built. The environment was very supportive for such work and the Labs provided a large pool of people with ideas and problems to challenge and test anything built. Thus I could write in [114]: "There never was a C++ paper design; design, documentation, and implementation went on simultaneously. Naturally, the C++ front-end is written in C++. There never was a "C++ project" either, or a "C++ design committee". Throughout, C++ evolved, and continues to evolve, to cope with problems encountered by users, and through discussions between the author and his friends and colleagues".

2.4 From C with Classes to C++

During 1982, it became clear to me that C with Classes was a "medium success" and would remain so until it died. The success of C with Classes was, I think, a simple consequence of meeting its design aim: C with Classes helped organize a large class of programs significantly better than C. Crucially, this was achieved without the loss of run-time efficiency and without requiring unacceptable cultural changes in development organizations. The factors limiting its success were partly the limited set of new facilities offered over C and partly the pre-processor technology used to implement C with Classes. C with Classes simply didn't provide support for people who were willing to invest significant effort to reap matching benefits: C with Classes was an important step in the right direction, but only one small step. As a result of this analysis, I began designing a cleaned-up and extended successor to C with Classes and implementing it using traditional compiler technology.

In the move from C with Classes to C++, the type checking was generally improved in ways that are possible only using a proper compiler front-end with full understanding of all syntax and semantics. This addressed a major problem with C with Classes. In addition, virtual functions, function name and operator overloading, references, constants (`const`), and many minor facilities were added. To many, virtual functions were the major addition, as they enable object-oriented programming. I had been unable to convince my colleagues of their utility, but saw them as essential for the support of a key programming style ("paradigm").

After a couple of years of use, release 2.0 was a major release providing a significantly expanded set of features, such as type-safe linkage, abstract classes, and multiple inheritance. Most of these extensions and refinements represented experience gained with C++ and could not have been added earlier without more foresight than I possessed.

2.5 Chronology

The chronology of the early years can be summarized:

1979 Work on C with Classes starts; first C with Classes use

1983 1st C++ implementation in use

1984 C++ named

1985 Cfront Release 1.0 (first commercial release); *The C++ Programming Language* (TC++PL) [112]

1986 1st commercial Cfront PC port (Cfront 1.1, Glockenspiel)

1987 1st GNU C++ release

1988 1st Oregon Software C++ release; 1st Zortech C++ release;

1989 Cfront Release 2.0; *The Annotated C++ Reference Manual* [35]; ANSI C++ committee (J16) founded (Washington, D.C.)

1990 1st ANSI X3J16 technical meeting (Somerset, New Jersey); templates accepted (Seattle, WA); exceptions accepted (Palo Alto, CA); 1st Borland C++ release

1991 1st ISO WG21 meeting (Lund, Sweden); Cfront Release 3.0 (including templates); *The C++ Programming Language (2nd edition)* [118]

On average, the number of C++ users doubled every 7.5 months from 1 in October 1979 to 400,000 in October of 1991 [121]. It is of course very hard to count users, but during the early years I had contacts with everyone who shipped compilers, libraries, books, etc., so I'm pretty confident of these numbers. They are also consistent with later numbers from IDC [61].

3. The C++ World in 1991

In 1991, the second edition of my *The C++ Programming Language* [118] was published to complement the 1989 language definition *The Annotated C++ Reference Manual* ("the ARM") [35]. Those two books set the standard for C++ implementation and to some extent for programming techniques for years to come. Thus, 1991 can be seen as the end of C++'s preparations for entry into the mainstream. From then on, consolidation was a major issue. That year, there were five C++ compilers to choose from (AT&T, Borland, GNU, Oregon, and Zortech) and three more were to appear in 1992 (IBM, DEC, and Microsoft). The October 1991 AT&T release 3.0 of Cfront (my original C++ compiler [121]) was the first to support templates. The DEC and IBM compilers both implemented templates and exceptions, but Microsoft's did not, thus seriously setting back efforts to encourage programmers to use a more modern style. The effort to standardize C++, which had begun in 1989, was officially converted into an international effort under the auspices of ISO. However, this made no practical difference as even the organizational meeting in 1989 had large non-US participation. The main difference is that we refer to ISO C++ rather than ANSI C++. The C++ programmer — and would-be C++ programmer — could now choose from among about 100 books of greatly varying aim, scope, and quality.

Basically, 1991 was an ordinary, successful year for the C++ community. It didn't particularly stand out from the years before or after. The reason to start our story here is that my HOPL-II paper [120] left off in 1991.

3.1 The ISO C++ Standards Process

For the C++ community, the ISO standards process is central: The C++ community has no other formal center, no other forum for driving language evolution, no other organization that cares for the language itself and for the community as a whole. C++ has no owner corporation that determines a path for the language, finances its development, and provides marketing. Therefore, the C++ standards committee became the place where language and standard library evolution is seriously considered. To a large extent, the 1991-2006 evolution of C++ was determined by what could be done by the volunteer individuals in the committee and how far the dreaded "design by committee" could be avoided.

The American National Standards Institute committee for C++ (ANSI J16) was founded in 1989; it now works under the auspices of INCITS (InterNational Committee for Information Technology Standards, a US organization). In 1991, it became part of an international effort under the auspices of ISO. Several other countries (such as France, Japan, and the UK) have their own national committees that hold their own meetings (in person or electronically) and send representatives to the ANSI/ISO meetings. Up until the final vote on the C++98 standard [63] in 1997, the committee met three times a year for week-long meetings. Now, it meets twice a year, but depends far more on electronic communications in between meetings. The committee tries to alternate meetings between the North American continent and elsewhere, mostly Europe.

The members of the J16 committee are volunteers who have to pay (about $800 a year) for the privilege of doing all the work. Consequently, most members represent companies that are willing to pay fees and travel expenses, but there is always a small number of people who pay their own way. Each company participating has one vote, just like each individual, so a company cannot stuff the committee with employees. People who represent their nations in the ISO (WG21) and participate in the national C++ panels pay or not according to their national standards organization rules. The J16 convener runs the technical sessions and does formal votes. The voting procedures are very democratic. First come one or more "straw votes" in a working group as part of the process to improve the proposal and gain consensus. Then come the more formal J16 votes in full committee where only accredited members vote. The final (ISO) votes are done on a per-nation basis. The aim is for consensus, defined as a massive majority, so that the resulting standard is good enough for everybody, if not necessarily exactly what any one member would have preferred. For the 1997/1998 final standards ballot, the ANSI vote was 43-0 and the ISO vote 22-0. We really did reach consensus, and even unanimity. I'm told that such clear votes are unusual.

The meetings tend to be quite collegial, though obviously there can be very tense moments when critical and controversial decisions must be made. For example, the debate leading up to the `export` vote strained the committee (§5.2). The collegiality has been essential for C++. The committee is the only really open forum where implementers and users from different — and often vigorously competing — organizations can meet and exchange views. Without a committee with a dense web of professional, personal, and organizational connections, C++ would have broken into a mess of feuding dialects. The number of people attending a meeting varies between 40 and 120. Obviously, the attendance increases when the meeting is in a location with many C++ programmers (e.g., Silicon Valley) or something major is being decided (e.g., should we adopt the STL?). At the Santa Cruz meeting in 1996, I counted 105 people in the room at the same time.

Most technical work is done by individuals between meetings or in working groups. These working groups are officially "ad hoc" and have no formal standing. However, some lasts for many years and serve as a focus for work and as the institutional memory of the committee. The main long-lived working groups are:

- *Core* — chairs: Andrew Kornig, Jose Lajorie, Bill Gibbons, Mike Miller, Steve Adamczyk.
- *Evolution* (formerly *extensions*) — chair: Bjarne Stroustrup.
- *Library* — chairs: Mike Vilot, Beman Dawes, Matt Austern, Howard Hinnant.

When work is particularly hectic, these groups split into sub-working-groups focussed on specific topics. The aim is to increase the degree of parallelism in the process and to better use the skills of the many people present.

The official "chair" of the whole committee whose primary job is the ensure that all formal rules are obeyed and to report back to SC22 is called the convener. The original convener was Steve Carter (BellCore). Later Sam Harbinson (Tartan Labs and Texas Instruments), Tom Plum (Plum Hall), and Herb Sutter (Microsoft) served.

The J16 chairman conducts the meeting when the whole committee is gathered. Dmitri Lenkov (Hewlett-Packard) was the original J16 chair and Steve Clamage (Sun) took over after him.

The draft standard text is maintained by the project editor. The original project editor was Jonathan Shopiro (AT&T). He was followed by Andrew Koenig (AT&T) whose served 1992-2004, after which Pete Becker (Dinkumware) took over "the pen".

The committee consists of individuals of diverse interests, concerns, and backgrounds. Some represent themselves, some represent giant corporations. Some use PCs, some use UNIX boxes, some use mainframes, etc. Some would like C++ to become more of an object-oriented language (according to a variety of definitions of "object-oriented"), others would have been more comfortable had ANSI C been the end point of C's evolution. Many have a background in C, some do not. Some have a background in standards work, many do not. Some have a computer science background, some do not. Most are programmers, some are not. Some are language lawyers, some are not. Some serve end-users, some are tools suppliers. Some are interested in large projects, some are not. Some are interested in C compatibility, some are not. It is hard to find a generalization that covers them all.

This diversity of backgrounds has been good for C++; only a very diverse group could represent the diverse interests of the C++ community. The end results — such as the 1998 standard and the Technical Reports (§6.1, §6.2) — are something that is good enough for everyone represented, rather than something that is ideal for any one subcommunity. However, the diversity and size of the membership do make constructive discussion difficult and slow at times. In particular, this very open process is vulnerable to disruption by individuals whose technical or personal level of maturity doesn't encourage them to understand or respect the views of others. Part of the consideration of a proposal is a process of education of the committee members. Some members have claimed — only partly in jest — that they attend to get that education. I also worry that the voice of C++ users (that is, programmers and designers of C++ applications) can be drowned by the voices of language lawyers, would-be language designers, standards bureaucrats, implementers, tool builders, etc.

To get an idea about what organizations are represented, here are some names from the 1991-2005 membership lists: Apple, AT&T, Bellcore, Borland, DEC, Dinkumware, Edison Design Group (EDG), Ericsson, Fujitsu, Hewlett-Packard, IBM, Indiana University, Los Alamos National Labs, Mentor Graphics, Microsoft, NEC, Object Design, Plum Hall, Siemens Nixdorf, Silicon Graphics, Sun Microsystems, Texas Instruments, and Zortech.

Changing the definition of a widely used language is very different from simple design from first principles. Whenever we have a "good idea", however major or minor, we must remember that

- there are hundreds of millions of lines of code "out there" — most will not be rewritten however much gain might result from a rewrite
- there are millions of programmers "out there" — most won't take out time to learn something new unless they consider it essential
- there are decade-old compilers still in use — many programmers can't use a language feature that doesn't compile on every platform they support
- there are many millions of outdated textbooks out there — many will still be in use in five years' time

The committee considers these factors and obviously that gives a somewhat conservative bias. Among other things, the members of the committee are indirectly responsible for well over 100 million lines of code (as representatives of their organizations). Many of the members are on the committee to promote change, but almost all do so with a great sense of caution and responsibility. Other members feel that their role is more along the lines of avoiding unnecessary and dangerous instability. Compatibility with previous versions of C++ (back to ARM C++ [35]), previous versions of C (back to K&R C [76]), and generations of corporate dialects is a serious issue for most members. We (the members of the committee) try to face future challenges, such as concurrency, but we do so remembering that C++ is at the base of many tool chains. Break C++ and the major implementations of Java and C# would also break. Obviously, the committee couldn't "break C++" by incompatibilities even if it wanted to. The industry would simply ignore a seriously incompatible standard and probably also start migrating away from C++.

3.2 Chronology

Looking forward beyond 1991, we can get some idea of the process by listing some significant decisions (votes):

1993 Run-time type identification accepted (Portland, Oregon) §5.1.2; namespaces accepted (Munich, Germany) §5.1.1

1994 `string` (templatized by character type) (San Diego, California)

1994 The STL (San Diego, California) §4.1

1996 `export` (Stockholm, Sweden) §5.2

1997 Final committee vote on the complete standard (Morristown, New Jersey)

1998 ISO C++ standard [63] ratified

2003 Technical Corrigendum ("mid-term bug-fix release") [66]; work on C++0x starts

2004 Performance technical report [67] §6.1; Library Technical Report (hash tables, regular expressions, smart pointers, etc.) [68] §6.2

2005 1st votes on features for C++0x (Lillehammer, Norway); `auto`, `static_assert`, and rvalue references accepted in principle; §8.3.2

2006 1st full committee (official) votes on features for C++0x (Berlin, Germany)

The city names reflect the location of the meeting where the decision was taken. They give a flavor of the participation. When the committee meets in a city, there are usually a dozen or more extra participants from nearby cities and countries. It is also common for the host to take advantage of the influx of C++ experts — many internationally known — to arrange talks to increase the understanding of C++ and its standard in the community. The first such arrangement was in Lund in 1991 when the Swedish delegation collaborated with Lund University to put on a two-day C++ symposium.

This list gives a hint of the main problem about the process from the point of view of someone trying to produce a coherent language and library, rather than a set of unrelated "neat features". At any time, the work focused on a number of weakly related specific topics, such as the definition of "undefined", whether it is possible to resume from an exception (it isn't), what functions should be provided for `string`, etc. It is extremely hard to get the committee to agree on an overall direction.

3.3 Why Change?

Looking at the list of decisions and remembering the committee's built-in conservative bias, the obvious question is: "Why change anything?" There are people who take the position that "standardization is to document existing practice". They were never more than a tiny fraction of the committee membership and represent an even smaller proportion of the vocal members of the C++ committee. Even people who say that they want "no change" ask for "just one or two improvements". In this, the C++ committee strongly resembles other language standardization groups.

Basically, we (the members of the committee) desire change because we hold the optimistic view that better language features and better libraries lead to better code. Here, "better" means something like "more maintainable", "easier to read", "catches more errors", "faster", "smaller", "more portable", etc. People's criteria differ, sometimes drastically. This view is optimistic because there is ample evidence that people can — and do — write really poor code in every language. However, most groups of programmers — including the C++ committee — are dominated by optimists. In particular, the conviction of a large majority of the C++ committee has consistently been that the quality of C++ code can be improved over the long haul by providing better language features and standard-library facilities. Doing so takes time: in some cases we may have to wait for a new generation of programmers to get educated. However, the committee is primarily driven by optimism and idealism — moderated by vast experience — rather than the cynical view of just giving people what they ask for or providing what "might sell".

An alternative view is that the world changes and a living language will change — whatever any committee says. So, we can work on improvements — or let others do it for us. As the world changes, C++ must evolve to meet new challenges. The most obvious alternative would not be "death" but capture by a corporation, as happened with Pascal and Objective C, which became Borland and Apple corporate languages, respectively.

After the Lund (Sweden) meeting in 1991, the following cautionary tale became popular in the C++ community:

> We often remind ourselves of the good ship Vasa. It was to be the pride of the Swedish navy and was built to be the biggest and most beautiful battleship ever. Unfortunately, to accommodate enough statues and guns, it underwent major redesigns and extension during construction. The result was that it only made it halfway across Stockholm harbor before a gust of wind blew it over and it sank, killing about 50 people. It has been raised and you can now see it in a museum in Stockholm. It is a beauty to behold — far more beautiful at the time than its unextended first design and far more beautiful today than if it had suffered the usual fate of a 17th century battleship — but that is no consolation to its designer, builders, and intended users.

This story is often recalled as a warning against adding features (that was the sense in which I told it to the committee). However, to complicate matters, there is another side to the story: Had the Vasa been completed as originally designed, it would have been sent to the bottom full of holes the first time it encountered a "modern two-deck" battleship. Ignoring changes in the world isn't an option (either).

4. The Standard Library: 1991-1998

After 1991, most major changes to the draft C++ standard were in the standard library. Compared to that, the language features were little changed even though we made an apparently infinite number of improvements to the text of the standard. To gain a perspective, note that the standard is 718 pages: 310 define the language and 366 define the standard library. The rest are appendices, etc. In addition, the C++ standard library includes the C standard library by reference;

that's another 81 pages. To compare, the base document for the standardization, the reference manual of TC++PL2 [118], contained 154 pages on the language and just one on the standard library.

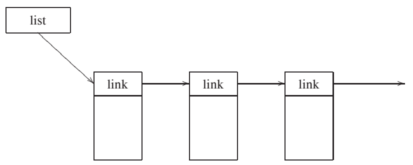
By far the most innovative component of the standard library is "the STL" — the containers, iterators, and algorithms part of the library. The STL influenced and continues to influence not only programming and design techniques but also the direction of new language features. Consequently, this is treated first here and in far greater detail than other standard library components.

4.1 The STL

The STL was the major innovation to become part of the standard and the starting point for much of the new thinking about programming techniques that have occurred since. Basically, the STL was a revolutionary departure from the way the C++ community had been thinking about containers and their use.

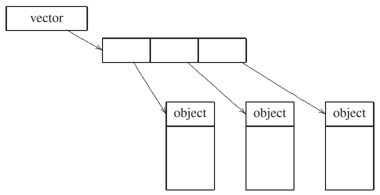
4.1.1 Pre-STL containers

From the earliest days of Simula, containers (such as lists) had been intrusive: An object could be put into a container if and only if its class had been (explicitly or implicitly) derived from a specific `Link` and/or `Object` class. This class contains the link information needed for management of objects in a container and provides a common type for elements. Basically, such a container is a container of references (pointers) to links/objects. We can graphically represent such a list like this:

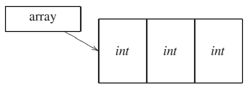


The links come from a base class `Link`.

Similarly, an "object-oriented" vector is a basically an array of references to objects. We can graphically represent such a vector like this:



The references to objects in the vector data structure point to objects of an `Object` base class. This implies that objects of a fundamental type, such as `int` and `double`, can't be put directly into containers and that the array type, which directly supports fundamental types, must be different from other containers:



Furthermore, objects of really simple classes, such as `complex` and `Point`, can't remain optimal in time and space if we want to put them into a container. In other words, Simula-style containers are intrusive, relying on data fields inserted into their element types, and provide indirect access to objects through pointers (references). Furthermore, Simula-style containers are not statically type safe. For example, a `Circle` may be added to a `list`, but when it is extracted we know only that it is an `Object` and need to apply a cast (explicit type conversion) to regain the static type.

Thus, Simula containers provide dissimilar treatment of built-in and user-defined types (only some of the latter can be in containers). Similarly, arrays are treated differently from user-defined containers (only arrays can hold fundamental types). This is in direct contrast to two of the clearest language-technical ideals for C++:

- Provide the same support for built-in and user-defined types
- What you don't use, you don't pay for (zero-overhead rule).

Smalltalk has the same fundamental approach to containers as Simula, though it makes the base class universal and thus implicit. The problems also appear later languages, such as Java and C# (though they – like Smalltalk – make use of a universal class and C# 2.0 applies C++-like specialization to optimize containers of integers). Many early C++ libraries (e.g. the NIHCL [50], early AT&T libraries [5]) also followed this model. It does have significant utility and many designers were familiar with it. However, I considered this double irregularity and the inefficiency (in time and space) that goes with it unacceptable for a truly general-purpose library (you can find a summary of my analysis in §16.2 of TC++PL3 [126]).

The lack of a solution to these logical and performance problems was the fundamental reason behind my "biggest mistake" of not providing a suitable standard library for C++ in 1985 (see D&E [121] §9.2.3): I didn't want to ship anything that couldn't directly handle built-in types as elements and wasn't statically type safe. Even the first "C with Classes" paper [108] struggled with that problem, unsuccessfully trying to solve it using macros. Furthermore, I specifically didn't want to provide something with covariant containers. Consider a `Vector` of some "universal" `Object` class:

```
void f(Vector& p)
{
    p[2] = new Pear;
}

void g()
```

```
{
   Vector apples(10);  // container of apples
   for(int i=0; i<10; ++i)
      apples[i] = new Apple;
   f(apples);
   // now apples contains a pear
}
```

The author of g pretends that apples is a Vector of Apples. However, Vector is a perfectly ordinary object-oriented container, so it really contains (pointers to) Objects. Since Pear is also an Object, f can without any problems put a Pear into it. It takes a run-time check (implicit or explicit) to catch the error/misconception. I remember how shocked I was when this problem was first explained to me (sometime in the early 1980s). It was just too shoddy. I was determined that no container written by me should give users such nasty surprises and significant run-time checking costs. In modern (i.e. post-1988) C++ the problem is solved using a container parameterized by the element type. In particular, Standard C++ offers vector<Apple> as a solution. Note that a vector<Apple> does not convert to a vector<Fruit> even when Apple implicitly converts to Fruit.

4.1.2 The STL emerges

In late 1993, I became aware of a new approach to containers and their use that had been developed by Alex Stepanov. The library he was building was called "The STL". Alex then worked at HP Labs but he had earlier worked for a couple of years at Bell Labs, where he had been close to Andrew Koenig and where I had discussed library design and template mechanisms with him. He had inspired me to work harder on generality and efficiency of some of the template mechanisms, but fortunately he failed to convince me to make templates more like Ada generics. Had he succeeded, he wouldn't have been able to design and implement the STL!

Alex showed the latest development in his decades-long research into generic programming techniques aiming for "the most general and most efficient code" based on a rigorous mathematical foundation. It was a framework of containers and algorithms. He first explained his ideas to Andrew, who after playing with the STL for a couple of days showed it to me. My first reaction was puzzlement. I found the STL style of containers and container use very odd, even ugly and verbose. For example, you sort a vector of doubles, vd, according to their absolute value like this:

```
sort(vd.begin(), vd.end(), Absolute<double>());
```

Here, vd.begin() and vd.end() specify the beginning and end of the vector's sequence of elements and Absolute<double>() compares absolute values.

The STL separates algorithms from storage in a way that's classical (as in math) but not object oriented. Furthermore, it separates policy decisions of algorithms, such as sorting criteria and search criteria, from the algorithm, the container, and the element class. The result is unsurpassed flexibility and — surprisingly — performance.

Like many programmers acquainted with object-oriented programming, I thought I knew roughly how code using containers had to look. I imagined something like Simula-style containers augmented by templates for static type safety and maybe abstract classes for iterator interfaces. The STL code looked very different. However, over the years I had developed a checklist of properties that I considered important for containers:

1. Individual containers are simple and efficient.
2. A container supplies its "natural" operations (e.g., list provides put and get and vector provides subscripting).
3. Simple operators, such as member access operations, do not require a function call for their implementation.
4. Common functionality can be provided (maybe through iterators or in a base class)
5. Containers are by default statically type-safe and homogeneous (that is, all elements in a container are of the same type).
6. A heterogeneous container can be provided as a homogeneous container of pointers to a common base.
7. Containers are non-intrusive (i.e., an object need not have a special base class or link field to be a member of a container).
8. A container can contain elements of built-in types
9. A container can contain structs with externally imposed layouts.
10. A container can be fitted into a general framework (of containers and operations on containers).
11. A container can be sorted without it having a sort member function.
12. "Common services" (such as persistence) can be provided in a single place for a set of containers (in a base class?).

A slightly differently phrased version can be found in [124].

To my amazement the STL met all but one of the criteria on that list! The missing criterion was the last. I had been thinking of using a common base class to provide services (such as persistence) for all derived classes (e.g., all objects or all containers). However, I didn't (and don't) consider such services intrinsic to the notion of a container. Interestingly, some "common services" can be expressed using "concepts" (§8.3.3) that specifically address the issue of what can be required of a set of types, so C++0x (§8) is likely to bring the STL containers even closer to the ideals expressed in that list.

It took me some time — weeks — to get comfortable with the STL. After that, I worried that it was too late to introduce a completely new style of library into the C++ community.

Considering the odds to get the standards committee to accept something new and revolutionary at such a late stage of the standards process, I decided (correctly) that those odds were very low. Even at best, the standard would be delayed by a year — and the C++ community urgently needed that standard. Also, the committee is fundamentally a conservative body and the STL was revolutionary.

So, the odds were poor, but I plodded on hoping. After all, I really did feel very bad about C++ not having a sufficiently large and sufficiently good standard library [120] (D&E [121] §9.2.3). Andrew Koenig did his best to build up my courage and Alex Stepanov lobbied Andy and me as best he knew how to. Fortunately, Alex didn't quite appreciate the difficulties of getting something major through the committee, so he was less daunted and worked on the technical aspects and on teaching Andrew and me. I began to explain the ideas behind the STL to others; for example, the examples in D&E §15.6.3.1 came from the STL and I quoted Alex Stepanov: "C++ is a powerful enough language — the first such language in our experience — to allow the construction of generic programming components that combine mathematical precision, beauty, and abstractness with the efficiency of non-generic hand-crafted code". That quote is about generic programming in general (§7.2.1) and the STL in particular.

Andrew Koenig and I invited Alex to give an evening presentation at the October 1993 standards committee meeting in San Jose, California: "It was entitled *The Science of C++ Programming* and dealt mostly with axioms of regular types — connecting construction, assignment and equality. I also described axioms of what is now called Forward Iterators. I did not at all mention any containers and only one algorithm: `find`". [105]. That talk was an audacious piece of rabble rousing that to my amazement and great pleasure basically swung the committee away from the attitude of "it's impossible to do something major at this stage" to "well, let's have a look".

That was the break we needed! Over the next four months, we (Alex, his colleague Meng Lee, Andrew, and I) experimented, argued, lobbied, taught, programmed, redesigned, and documented so that Alex was able to present a complete description of the STL to the committee at the March 1994 meeting in San Diego, California. Alex arranged a meeting for C++ library implementers at HP later in 1994. The participants were Meng Lee (HP), Larry Podmolik, Tom Keffer (Rogue Wave), Nathan Myers, Mike Vilot, Alex, and I. We agreed on many principles and details, but the size of the STL emerged as the major obstacle. There was no consensus about the need for large parts of the STL, there was a (realistic) worry that the committee wouldn't have the time to properly review and more formally specify something that large, and people were simply daunted by the sheer number of things to understand, implement, document, teach, etc. Finally, at Alex's urging, I took a pen and literally crossed out something like two thirds of all the text. For each facility, I challenged Alex and the other library experts to explain — very briefly — why it couldn't be cut and why it would benefit most C++ programmers. It was a horrendous exercise. Alex later claimed that it broke his heart. However, what emerged from that slashing is what is now known as the STL [103] and it made it into the ISO C++ standard at the October 1994 meeting in Waterloo, Canada — something that the original and complete STL would never have done. Even the necessary revisions of the "reduced STL" delayed the standard by more than a year. The worries about size and complexity (even after my cuts) were particularly acute among library implementers concerned about the cost of providing a quality implementation. For example, I remember Roland Hartinger (representing Siemens and Germany) worrying that acceptance of the STL would cost his department one million marks. In retrospect, I think that I did less damage than we had any right to hope for.

Among all the discussions about the possible adoption of the STL one memory stands out: Beman Dawes calmly explaining to the committee that he had thought the STL too complex for ordinary programmers, but as an exercise he had implemented about 10% of it himself so he no longer considered it beyond the standard. Beman was (and is) one of the all too rare application builders in the committee. Unfortunately, the committee tends to be dominated by compiler, library, and tools builders.

I credit Alex Stepanov with the STL. He worked with the fundamental ideals and techniques for well over a decade before the STL, unsuccessfully using languages such as Scheme and Ada [101]. However, Alex is always the first to insist that others took part in that quest. David Musser (a professor at Rensselaer Polytechnic Institute) has been working with Alex on generic programming for almost two decades and Meng Lee worked closely with him at HP helping to program the original STL. Email discussions between Alex and Andrew Koenig also helped. Apart from the slashing exercise, my technical contributions were minor. I suggested that various information related to memory be collected into a single object — what became the allocators. I also drew up the initial requirement tables on Alex's blackboard, thus creating the form in which the standard specifies the requirements that STL templates place on their arguments. These requirements tables are actually an indicator that the language is insufficiently expressive — such requirements should be part of the code; see §8.3.3.

Alex named his containers, iterators, and algorithm library "the STL". Usually, that's considered an acronym for "Standard Template Library". However, the library existed — with that name — long before it was any kind of standard and most parts of the standard library rely on templates. Wits have suggested "STepanov and Lee" as an alternative explanation, but let's give Alex the final word: " 'STL' stands for

'STL' ". As in many other cases, an acronym has taken on a life of its own.

4.1.3 STL ideals and concepts

So what is the STL? It comes from an attempt to apply the mathematical ideal of generality to the problem of data and algorithms. Consider the problem of storing objects in containers and writing algorithms to manipulate such objects. Consider this problem in the light of the ideals of direct, independent, and composable representation of concepts:

- express concepts directly in code
- express relations among concepts directly in code
- express independent concepts in independent code
- compose code representing concepts freely wherever the composition makes sense

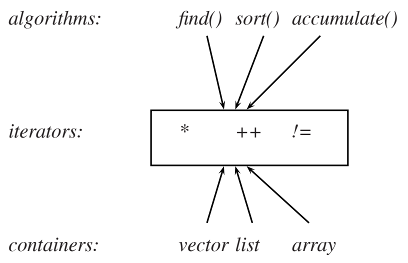
We want to be able to

- store objects of a variety of types (e.g. `ints`, `Points`, and pointers to `Shapes`)
- store those objects in a variety of containers (e.g. `list`, `vector`, and `map`),
- apply a variety of algorithms (e.g. `sort`, `find`, and `accumulate`) to the objects in the containers, and
- use a variety of criteria (comparisons, predicates, etc.) for those algorithms

Furthermore, we want the use of these objects, containers, and algorithms to be statically type safe, as fast as possible, as compact as possible, not verbose, and readable. Achieving all of this simultaneously is not easy. In fact, I spent more than ten years unsuccessfully looking for a solution to this puzzle (§4.1.2).

The STL solution is based on parameterizing containers with their element types and on completely separating the algorithms from the containers. Each type of container provides an iterator type and all access to the elements of the container can be done using only iterators of that type. The iterator defines the interface between the algorithm and the data on which it operates. That way, an algorithm can be written to use iterators without having to know about the container that supplied them. Each type of iterator is completely independent of all others except for supplying the same semantics to required operations, such as * and ++.

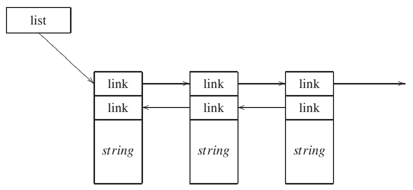
Algorithms use iterators and container implementers implement iterators for their containers:



Let's consider a fairly well-known example, the one that Alex Stepanov initially showed to the committee (San Jose, California, 1993). We want to find elements of various types in various containers. First, here are a couple of containers:

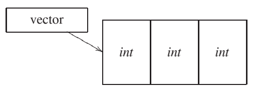
```
vector<int> vi;    // vector of ints
list<string> ls;   // list of strings
```

The `vector` and `list` are the standard library versions of the notions of vector and list implemented as templates. An STL container is a non-intrusive data structure into which you can copy elements of any type. We can graphically represent a (doubly linked) `list<string>` like this:



Note that the link information is *not* part of the element type. An STL container (here, `list`) manages the memory for its elements (here, `strings`) and supplies the link information.

Similarly, we can represent a `vector<int>` like this:

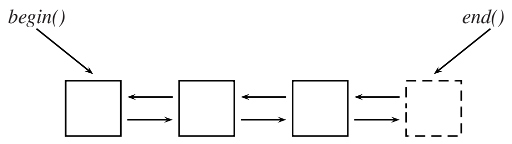


Note that the elements are stored in memory managed by the `vector` and `list`. This differs from the Simula-style containers in §4.1.1 in that it minimizes allocation operations, minimizes per-object memory, and saves an indirection on each access to an element. The corresponding cost is a copy operation when an object is first entered into a container; if a copy is expensive, programmers tend to use pointers as elements.

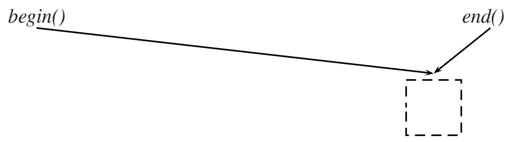
Assume that the containers `vi` and `ls` have been suitably initialized with values of their respective element types. It then makes sense to try to find the first element with the value `777` in `vi` and the first element with the value `"Stepanov"` in `ls`:

```
vector<int>::iterator p
     = find(vi.begin(),vi.end(),777);
list<string>::iterator q
     = find(ls.begin(),ls.end(),"Stepanov");
```

The basic idea is that you can consider the elements of any container as a sequence of elements. A container "knows" where its first element is and where its last element is. We call an object that points to an element "an iterator". We can then represent the elements of a container by a pair of iterators, `begin()` and `end()`, where `begin()` points to the first element and `end()` to one-beyond-the-last element. We can represent this general model graphically:



The `end()` iterator points to one-past-the-last element rather than to the last element to allow the empty sequence not to be a special case:



What can you do with an iterator? You can get the value of the element pointed to (using `*` just as with a pointer), make the iterator point to the next element (using `++` just as with a pointer) and compare two iterators to see if they point to the same element (using `==` or `!=` of course). Surprisingly, this is sufficient for implementing `find()`:

```
template<class Iter, class T>
Iter find(Iter first, Iter last, const T& val)
{
   while (first!=last && *first!=val)
      ++first;
   return first;
}
```

This is a simple — very simple, really — function template. People familiar with C and C++ pointers should find the code easy the read: `first!=last` checks whether we reached the end and `*first!=val` checks whether we found the value that we were looking for (`val`). If not, we increment the iterator `first` to make it point to the next element and try again. Thus, when `find()` returns, its value will point to either the first element with the value `val` or one-past-the-last element (`end()`). So we can write:

```
vector<int>::iterator p =
   find(vi.begin(),vi.end(),7);

if (p != vi.end()) {  // we found 7
   // ...
}
else {                // no 7 in vi
   // ...
}
```

This is very, very simple. It is simple like the first couple of pages in a math book and simple enough to be really fast. However, I know that I wasn't the only person to take significant time figuring out what really is going on here and longer to figure out why this is actually a good idea. Like simple math, the first STL rules and principles generalize beyond belief.

Consider first the implementation: In the call `find(vi.begin(),vi.end(),7)`, the iterators `vi.begin()` and `vi.end()` become `first` and `last`, respectively, inside. To `find()`, `first` is simply "something that points to an `int`". The obvious implementation of `vector<int>::iterator` is therefore a pointer to `int`, an `int*`. With that implementation, `*` becomes pointer dereference, `++` becomes pointer increment, and `!=` becomes pointer comparison. That is, the implementation of `find()` is obvious and optimal.

Please note that the STL does not use function calls to access the operations (such as `*` and `!=`) that are effectively arguments to the algorithm because they depend on a template argument. In this, templates differ radically from most mechanisms for "generics", relying on indirect function calls (like virtual functions), as provided by Java and C#. Given a good optimizer, `vector<int>::iterator` can without overhead be a class with `*` and `++` provided as inline functions. Such optimizers are now not uncommon and using a iterator class rather than a pointer improves type checking by catching unwarranted assumptions, such as that the `iterator` for a `vector` is a pointer:

```
int* p = find(vi.begin(),vi.end(),7); // error

// verbose, but correct:
vector<int>::iterator q =
   find(vi.begin(),vi.end(),7);
```

C++0x will provide ways of dealing with the verbosity; see §8.3.2.

In addition, *not* defining the interface between an algorithm and its type arguments as a set of functions with unique types provides a degree of flexibility that proved very important [130] (§8.3.3). For example, the standard library algorithm `copy` can copy between different container types:

```
void f(list<int>& lst, vector<int>& v)
{
     copy(lst.begin(), lst.end(), v.begin());
     // ...
```

```
        copy(v.begin(), v.end(), lst.end());
    }
```

So why didn't we just dispense with all that "iterator stuff" and use pointers? One reason is that `vector<int>::iterator` could have been a class providing range checked access. For a less subtle explanation, have a look at another call of `find()`:

```
list<string>::iterator q =
   find(ls.begin(),ls.end(),"McIlroy");

if (q != ls.end()) {  // we found "McIlroy"
   // ...
}
else {                // no "McIlroy" in ls
   // ...
}
```

Here, `list<string>::iterator` isn't going to be a `string*`. In fact, assuming the most common implementation of a linked list, `list<string>::iterator` is going to be a `Link<string>*` where `Link` is a link node type, such as:

```
template<class T> struct Link {
   T value;
   Link* suc;
   Link* pre;
};
```

That implies that * means `p->value` ("return the value field"), ++ means `p->suc` ("return a pointer to the next link"), and != pointer comparison (comparing `Link*`s). Again the implementation is obvious and optimal. However, it is completely different from what we saw for `vector<int>::iterator`.

We used a combination of templates and overload resolution to pick radically different, yet optimal, implementations of operations used in the definition of `find()` for each use of `find()`. Note that there is no run-time dispatch, no virtual function calls. In fact, there are only calls of trivially inlined functions and fundamental operations, such as * and ++ for a pointer. In terms of execution time and code size, we have hit the absolute minimum!

Why not use "sequence" or "container" as the fundamental notion rather than "pair of iterators"? Part of the reason is that "pair of iterators" is simply a more general concept than "container". For example, given iterators, we can sort the first half of a container only: `sort(vi.begin(), vi.begin()+vi.size()/2)`. Another reason is that the STL follows the C++ design rules that we must provide transition paths and support built-in and user-defined types uniformly. What if someone kept data in an ordinary array? We can still use the STL algorithms. For example:

```
int buf[max];
// ... fill buf ...
int*  p = find(buf,buf+max,7);

if (p != buf+max) {  // we found 7
   // ...
}
else {               // no 7 in buf
   // ...
}
```

Here, the `*`, `++`, and `!=` in `find()` really are pointer operations! Like C++ itself, the STL is compatible with older notions such as C arrays. Thus, the STL meets the C++ ideal of always providing a transition path (§2). It also meets the ideal of providing uniform treatment to user-defined types (such as `vector`) and built-in types (in this case, array) (§2).

Another reason for basing algorithms on iterators, rather than on containers or an explicit sequence abstraction, was the desire for optimal performance: using iterators directly rather than retrieving a pointer or an index from another abstraction eliminates a level of indirection.

As adopted as the containers and algorithms framework of the ISO C++ standard library, the STL consists of a dozen containers (such as `vector`, `list`, and `map`) and data structures (such as arrays) that can be used as sequences. In addition, there are about 60 algorithms (such as `find`, `sort`, `accumulate`, and `merge`). It would not be reasonable to present all of those here. For details, see [6, 126].

So, we can use simple arithmetic to see how the STL technique of separating algorithms from containers reduces the amount of source code we have to write and maintain. There are 60*12 (that is, 720) combinations of algorithm and container in the standard but just 60+12 (that is, 72) definitions. The separation reduces the combinatorial explosion to a simple addition. If we consider element types and policy parameters (function objects, see §4.1.4) for algorithms we see an even more impressive gain: Assume that we have N algorithms with M alternative criteria (policies) and X containers with Y element types. Then, the STL approach gives us N+M+X+Y definitions whereas "hand-crafted code" requires N*M*X*Y definitions. In real designs, the difference isn't quite that dramatic because typically designers attack that huge N*M*X*Y figure with a combination of conversions (one container to another, one data type to another), class derivations, function parameters, etc., but the STL approach is far cleaner and more systematic than earlier alternatives.

The key to both the elegance and the performance of the STL is that it — like C++ itself — is based directly on the hardware model of memory and computation. The STL notion of a sequence is basically that of the hardware's view of memory as a set of sequences of objects. The basic semantics of the STL map directly into hardware instructions allowing algorithms to be implemented optimally. The compile-time resolution of templates and the perfect inlining they support is then key to the efficient mapping of the high-level expression using the STL to the hardware level.

4.1.4 Function objects

The STL and generic programming in general owes a — freely and often acknowledged (e.g., [124]) — debt to functional programming. So where are the lambdas and higher-order functions? C++ doesn't directly support anything like that (though there are always proposals for nested functions, closures, lambdas, etc.; see §8.2). Instead, classes that define the application operator, called function objects (or even "functors"), take that role and have become the main mechanism of parameterization in modern C++. Function objects build on general C++ mechanisms to provide unprecedented flexibility and performance.

The STL framework, as described so far, is somewhat rigid. Each algorithm does exactly one thing in exactly the way the standard specifies it to. For example, using `find()`, we find an element that is equal to the value we give as the argument. It is actually more common to look for an element that has some desired property, such as matching strings without case sensitivity or matching floating-point values allowing for very slight differences.

As an example, instead of finding a value 7, let's look for a value that meets some predicate, say, being less than 7:

```
vector<int>::iterator p =
   find_if(v.begin(),v.end(),Less_than<int>(7));

if (p != vi.end()) { // element < 7
   // ...
}
else {                // no such element
   // ...
}
```

What is `Less_than<int>(7)`? It is a function object; that is, it is an object of a class that has the application operator, `()`, defined to perform an action:

```
template<class T> struct Less_than {
   T value;
   Less_than(const T& v) :value(v) { }
   bool operator()(const T& v) const
      { return v<value; }
};
```

For example:

```
Less_than<double> f(3.14); // f holds 3.14
bool b1 = f(3);  // true: 3<3.14 is true
bool b2 = f(4);  // false: 4<3.14 is false
```

From the vantage point of 2005, it seems odd that function objects are not mentioned in D&E or TC++PL1. They deserve a whole section. Even the use of a user-defined application operator, `()`, isn't mentioned even though it has had a long and distinguished career. For example, it was among the initial set of operators (after =; see D&E §3.6) that I allowed to be overloaded and was among many other things used to mimic Fortran subscript notation [112].

We used the STL algorithm `find_if` to apply `Less_than<int>(7)` to the elements of the `vector`. The definition of `find_if` differs from `find()`'s definition in using a user-supplied predicate rather than equality:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
   while (first!=last && !pred(*first))
      ++first;
   return first;
}
```

We simply replaced `*first!=val` with `!pred(*first)`. The function template `find_if()` will accept any object that can be called given an element value as its argument. In particular, we could call `find_if()` with an ordinary function as its third argument:

```
bool less_than_7(int a)
{
    return a<7;
}

vector<int>::iterator p =
   find_if(v.begin(),v.end(),less_than_7);
```

However, this example shows why we often prefer a function object over a function: The function object can be initialized with one (or more) arguments and carry information along for later use. A function object can carry a state. That makes for more general and more elegant code. If needed, we can also examine that state later. For example:

```
template<class T>
struct Accumulator {  // keep the sum of n values
   T value;
   int count;
   Accumulator() :value(), count(0) { }
   Accumulator(const T& v) :value(v), count(0) { }
   void operator()(const T& v)
      { ++count; value+=v; }
};
```

An `Accumulator` object can be passed to an algorithm that calls it repeatedly. The partial result is carried along in the object. For example:

```
int main()
{
   vector<double> v;
   double d;
   while (cin>>d) v.push_back(d);

   Accumulator<double> ad;
   ad = for_each(v.begin(),v.end(), ad);
   cout << "sum==" << ad.value
        << ", mean==" << ad.value/ad.count
        << '\n';
     return 0;
}
```

The standard library algorithm for_each simply applies its third argument to each element of its sequence and returns that argument as its return value. The alternative to using a function object would be a messy use of global variables to hold value and count. In a multithreaded system, such use of global variables is not just messy, but gives incorrect results.

Interestingly, simple function objects tend to perform better than their function equivalents. The reason is that they tend to be simple classes without virtual functions, so that when we call a member function the compiler knows exactly which function we are calling. That way, even a simple-minded compiler has all the information needed to inline. On the other hand, a function used as a parameter is passed as a pointer, and optimizers have traditionally been incapable of performing optimizations involving pointers. This can be very significant (e.g. a factor of 50 in speed) when we pass an object or function that performs a really simple operation, such as the comparison criteria for a sort. In particular, inlining of function objects is the reason that the STL (C++ standard library) sort() outperforms the conventional qsort() by several factors when sorting arrays of types with simple comparison operators (such as int and double) [125]. Spurred on by the success of function objects, some compilers now can do inlining for pointers to functions as long as a constant is used in the call. For example, today, some compilers can inline calls to compare from within qsort:

```
bool compare(double* a, double* b) { /* ... */ }
// ...
qsort(p,max,sizeof(double),compare);
```

In 1994, no production C or C++ compiler could do that.

Function objects are the C++ mechanism for higher-order constructs. It is not the most elegant expression of high-order ideas, but it is surprisingly expressive and inherently efficient in the context of a general purpose language. To get the same efficiency of code (in time and space) from a general implementation of conventional functional programming facilities requires significant maturity from an optimizer. As an example of expressiveness, Jaakko Järvi and Gary Powell showed how to provide and use a lambda class that made the following example legal with its obvious meaning [72]:

```
list<int> lst;
// ...
Lambda x;
list<int>::iterator p =
   find_if(lst.begin(),lst.end(),x<7);
```

Note how overload resolution enables us to make the element type, int, implicit (deduced). If you want just < to work, rather than building a general library, you can add definitions for Lambda and < in less than a dozen lines of code. Using Less_than from the example above, we can simply write:

```
class Lambda {};

template<class T>
Less_than<T> operator<(Lambda,const T& v)
{
   return Less_than<T>(v);
}
```

So, the argument x<7 in the call of find_if becomes a call of operator<(Lambda,const int&), which generates a Less_than<int> object. That's exactly what we used explicitly in the first example in this section. The difference here is just that we have achieved a much simpler and more intuitive syntax. This is a good example of the expressive power of C++ and of how the interface to a library can be simpler than its implementation. Naturally, there is no run-time or space overhead compared to a laboriously written loop to look for an element with a value less than 7.

The closest that C++ comes to higher-order functions is a function template that returns a function object, such as operator< returning a Less_than of the appropriate type and value. Several libraries have expanded that idea into quite comprehensive support for functional programming (e.g., Boost's Function objects and higher-order programming libraries [16] and FC++ [99]).

4.1.5 Traits

C++ doesn't offer a general compile-time way of asking for properties of a type. In the STL, and in many other libraries using templates to provide generic type-safe facilities for diverse types, this became a problem. Initially, the STL used overloading to deal with this (e.g., note the way the type int is deduced and used in x<7; §4.1.4). However, that use of overloading was unsystematic and therefore unduly difficult and error prone. The basic solution was discovered by Nathan Myers during the effort to templatize iostream and string [88]. The basic idea is to provide an auxiliary template, "a trait", to contain the desired information about a set of types. Consider how to find the type of the elements pointed to by an iterator. For a list_iterator<T> it is list_iterator<T>::value_type and for an ordinary pointer T* it is T. We can express that like this:

```
template<class Iter>
struct iterator_trait {
   typedef Iter::value_type value_type;
};

template<class T>
struct iterator_trait<T*> {
   typedef T value_type;
};
```

That is, the value_type of an iterator is its member type value_type. However, pointers are a common form of iterators and they don't have any member types. So, for pointers we use the type pointed to as value_type. The language construct involved is called partial specialization (added to C++ in 1995; §5). Traits can lead to somewhat bloated source code (though they have no object code or run-time cost) and even though the technique is very extensible it often requires explicit programmer attention when a new type is added. Traits are now ubiquitous in template-based C++ libraries. However, the "concepts" mechanism (§8.3.3) promises to make many traits redundant by providing direct language support for the idea of expressing properties of types.

4.1.6 Iterator categories

The result of the STL model as described so far could easily have become a mess with each algorithm depending on the peculiarities of the iterators offered by specific containers. To achieve interoperability, iterator interfaces have to be standardized. It would have been simplest to define a single set of operators for every iterator. However, to do so would have been doing violence to reality: From an algorithmic point of view lists, vectors, and output streams really do have different essential properties. For example, you can efficiently subscript a vector, you can add an element to a list without disturbing neighboring elements, and you can read from an input stream but not from an output stream. Consequently the STL provides a classification of iterators:

- input iterator (ideal for homogeneous stream input)
- output iterator (ideal for homogeneous stream output)
- forward iterator (we can read and write an element repeatedly, ideal for singly-linked lists)
- bidirectional iterator (ideal for doubly-linked lists)
- random access iterator (ideal for vectors and arrays)

This classification acts as a guide to programmers who care about interoperability of algorithms and containers. It allows us to minimize the coupling of algorithms and containers. Where different algorithms exist for different iterator categories, the most suitable algorithm is automatically chosen through overloading (at compile time).

4.1.7 Complexity requirements

The STL included complexity measures (using the big-O notation) for every standard library operation and algorithm. This was novel in the context of a foundation library for a language in major industrial use. The hope was and still is that this would set a precedent for better specification of libraries. Another — less innovative — aspect of this is a fairly systematic use of preconditions and postconditions in the specification of the library.

4.1.8 Stepanov's view

The description of the STL here is (naturally) focused on language and library issues in the context of C++. To get a complementary view, I asked Alexander Stepanov for his perspective [106]:

In October of 1976 I observed that a certain algorithm — parallel reduction — was associated with monoids: collections of elements with an associative operation. That observation led me to believe that it is possible to associate every useful algorithm with a mathematical theory and that such association allows for both widest possible use and meaningful taxonomy. As mathematicians learned to lift theorems into their most general settings, so I wanted to lift algorithms and data structures. One seldom needs to know the exact type of data on which an algorithm works since most algorithms work on many similar types. In order to write an algorithm one needs only to know the properties of operations on data. I call a collection of types with similar properties on which an algorithm makes sense the *underlying concept* of the algorithm. Also, in order to pick an efficient algorithm one needs to know the complexity of these operations. In other words, complexity is an essential part of the interface to a concept.

In the late '70s I became aware of John Backus's work on FP [7]. While his idea of programming with functional forms struck me as essential, I realized that his attempt to permanently fix the number of functional forms was fundamentally wrong. The number of functional forms — or, as I call them now, generic algorithms — is always growing as we discover new algorithms. In 1980 together with Dave Musser and Deepak Kapur I started working on a language Tecton to describe algorithms defined on algebraic theories. The language was functional since I did not realize at the time that memory and pointers were a fundamental part of programming. I also spent time studying Aristotle and his successors which led me to better understanding of fundamental operations on objects like equality and copying and the relation between whole and part.

In 1984 I started collaborating with Aaron Kershenbaum who was an expert on graph algorithms. He was able to convince me to take arrays seriously. I viewed sequences as recursively definable since it was commonly perceived to be the "theoretically sound" approach. Aaron showed me that many fundamental algorithms depended on random access. We produced a large set of components in Scheme and were able to implement generically some complicated graph algorithms.

The Scheme work led to a grant to produce a generic library in Ada. Dave Musser and I produced a generic library that dealt with linked structures. My attempts to implement algorithms that work on any sequential structure (both lists and arrays) failed because of the state of Ada compilers at the time. I had equivalences to many STL algorithms, but could not compile them.

Based on this work, Dave Musser and I published a paper where we introduced the notion of generic programming insisting on deriving abstraction from useful efficient algorithms. The most important thing I learned from Ada was the value of static typing as a design tool. Bjarne Stroustrup had learned the same lesson from Simula.

In 1987 at Bell Labs Andy Koenig taught me semantics of C. The abstract machine behind C was a revelation. I also read lots of UNIX and Plan 9 code: Ken Thompson's and Rob Pike's code certainly influenced STL. In any case, in 1987 C++ was not ready for STL and I had to move on.

At that time I discovered the works of Euler and my perception of the nature of mathematics underwent a dramatic transformation. I was de-Bourbakized, stopped believing in sets, and was expelled from the Cantorian paradise. I still believe in abstraction, but now I know that one ends with abstraction, not starts with it. I learned that one has to adapt abstractions to reality and not the other way around. Mathematics stopped being a science of theories but reappeared to me as a science of numbers and shapes.

In 1993, after five years working on unrelated projects, I returned to generic programming. Andy Koenig suggested that I write a proposal for including my library into the C++ standard, Bjarne Stroustrup enthusiastically endorsed the proposal and in less than a year STL was accepted into the standard. STL is the result of 20 years of thinking but of less than two years of funding.

STL is only a limited success. While it became a widely used library, its central intuition did not get across. People confuse generic programming with using (and abusing) C++ templates. Generic programming is about abstracting and classifying algorithms and data structures. It gets its inspiration from Knuth and not from type theory. Its goal is the incremental construction of systematic catalogs of useful, efficient and abstract algorithms and data structures. Such an undertaking is still a dream.

You can find references to the work leading to STL at `www.stepanovpapers.com`.

I am more optimistic about the long-term impact of Alex's ideas than he is. However, we agree that the STL is just the first step of a long journey.

4.1.9 The impact of the STL

The impact of the STL on the thinking on C++ has been immense. Before the STL, I consistently listed three fundamental programming styles ("paradigms") as being supported by C++ [113]:

- Procedural programming
- Data abstraction
- Object-oriented programming

I saw templates as support for data abstraction. After playing with the STL for a while, I factored out a fourth style:

- Generic programming

The techniques based on the use of templates and largely inspired by techniques from functional programming are qualitatively different from traditional data abstraction. People simply think differently about types, objects, and resources. New C++ libraries are written — using templates — to be statically type safe and efficient. Templates are the key to embedded systems programming and high-performance numeric programming where resource management and correctness are key [67]. The STL itself is not always ideal in those areas. For example, it doesn't provide direct support for linear algebra and it can be tricky to use in hard-real-time systems where free store use is banned. However, the STL demonstrates what can be done with templates and gives examples of effective techniques. For example, the use of iterators (and allocators) to separate logical memory access from actual memory access is key to many high-performance numeric techniques [86, 96] and the use of small, easily inlined, objects is key to examples of optimal use of hardware in embedded systems programming. Some of these techniques are documented in the standards committee's technical report on performance (§6.1). The emphasis on the STL and on generic programming in the C++ community in the late 1990s and early 2000s is to a large extent a reaction — and a constructive alternative — to a trend in the larger software-development community towards overuse of "object-oriented" techniques relying excessively on class hierarchies and virtual functions.

Obviously, the STL isn't perfect. There is no one "thing" to be perfect relative to. However, it broke new ground and has had impact even beyond the huge C++ community (§9.3). It also inspired many to use templates both in more disciplined and more adventurous ways. People talk about "template meta-programming" (§7.2.2) and generative programming [31] and try to push the techniques pioneered by the STL beyond the STL. Another line of attack is to consider how C++ could better support effective uses of templates (concepts, `auto`, etc.; see §8).

Inevitably, the success of STL brought its own problems. People wanted to write all kinds of code in the STL style. However, like any other style or technique, the STL style or even generic programming in general isn't ideal for every kind of problem. For example, generic programming relying on templates and overloading completely resolves all name bindings at compile time. It does not provide a mechanism for bindings that are resolved at run time; that's what class hierarchies and their associated object-oriented design techniques are for. Like all successful language mechanisms and programming techniques, templates and generic programming became fashionable and severely overused. Programmers built truly baroque and brittle constructs based on the fact that template instantiation and deduction is Turing complete. As I earlier observed for C++'s object-oriented facilities and techniques: "Just because you can do it, doesn't mean that you have to". Developing a comprehensive and simple framework for using the different programming styles supported by C++ is a major challenge for the next few years. As a style of programming, "multi-paradigm programming" [121] is underdeveloped. It often provides solutions that are more elegant and outperforms alternatives [28], but we don't (yet) have a simple and systematic way of combining the programming styles. Even its name gives away its fundamental weakness.

Another problem with the STL is that its containers are non-intrusive. From the point of view of code clarity and independence of concepts, being non-intrusive is a huge advantage. However, it does mean that we need to copy elements into containers or insert objects with default values into containers and later give them the desired values. Sometimes that's inefficient or inconvenient. For example, people tend not to insert large objects into a `vector` for that reason; instead, they insert pointers to such large objects. Similarly, the implicit memory management that the standard containers provide for their elements is a major convenience, but there are applications (e.g., in some embedded and high-performance systems) where such implicit memory management must be avoided. The standard containers provide features for ensuring that (e.g., `reserve`), but they have to be understood and used to avoid problems.

4.2 Other Parts of the Standard Library

From 1994 onwards, the STL dominated the work on the standard library and provided its major area of innovation. However, it was not the only area of work. In fact, the standard library provides several components:

- basic language run-time support (memory management, run-time type information (RTTI), exceptions, etc.)
- the C standard library
- the STL (containers, algorithms, iterators, function objects)
- `iostreams` (templatized on character type and implicitly on locale)
- `locales` (objects characterizing cultural preferences in I/O)
- `string` (templatized on character type)
- `bitset` (a set of bits with logical operations)
- `complex` (templatized on scalar type)
- `valarray` (templatized on scalar type)
- `auto_ptr` (a resource handle for objects templatized on type)

For a variety of reasons, the stories of the other library components are less interesting and less edifying than the story of the STL. Most of the time, work on each of these components progressed in isolation from work on the others. There was no overall design or design philosophy. For example, `bitset` is range checked whereas `string` isn't. Furthermore, the design of several components (such as `string`, `complex`, and `iostream`) was constrained by compatibility concerns. Several (notably `iostream` and `locale`) suffered from the "second-system effect" as their designers tried to cope with all kinds of demands, constraints, and existing practice. Basically, the committee failed to contain "design by committee" so whereas the STL reflects a clear philosophy and coherent style, most of the other components suffered. Each represents its own style and philosophy, and some (such as `string`) manage simultaneously to present several. I think `complex` is the exception here. It is basically my original design [91] templatized to allow for a variety of scalar types:

```
complex<double> z;     // double-precision
complex<float> x;      // single-precision
complex<short> point;  // integer grid
```

It is hard to seriously mess up math.

The committee did have some serious discussions about the scope of the standard library. The context for this discussion was the small and universally accepted C standard library (which the C++ standard adopted with only the tiniest of modification) and the huge corporate foundation libraries. During the early years of the standards process, I articulated a set of guidelines for the scope of the C++ standard library:

> First of all, the key libraries now in almost universal use must be standardized. This means that the exact interface between C++ and the C standard libraries must be specified and the iostreams library must be specified. In addition, the basic language support must be specified. ...
>
> Next, the committee must see if it can respond to the common demand for "more useful and standard classes," such as `string`, without getting into a mess of design by committee and without competing with the C++ library industry. Any libraries beyond the C libraries and iostreams accepted by the committee must be in the nature of building blocks rather than more ambitious frameworks. The key role of a standard library is to ease communication between separately developed, more ambitious libraries.

The last sentence delineated the scope of the committee's efforts. The elaboration of the requirement for the standard library to consist of building blocks for more ambitious libraries and frameworks emphasized absolute efficiency and extreme generality. One example I frequently used to illustrate the seriousness of those demands was that a container where element access involved a virtual function call could not be sufficiently efficient and that a container that couldn't hold an arbitrary type could not be sufficiently general (see also §4.1.1). The committee felt that the role of the standard library was to support rather than supplant other libraries.

Towards the end of the work on the 1998 standard, there was a general feeling in the committee that we hadn't done enough about libraries, and also that there had been insufficient experimentation and too little attention to performance measurement for the libraries we did approve. The question was how to address those issues in the future. One — classical standards process — approach was to work on technical reports (see §6.2). Another was initiated by Beman Dawes in 1998, called Boost [16]. To quote from boost.org (August 2006):

> "Boost provides free peer-reviewed portable C++ source libraries. We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The Boost license encourages both commercial and non-commercial use. We aim to establish "existing practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization. Ten Boost libraries are already included in the C++ Standards Committee's Library Technical Report (TR1) as a step toward becoming part of a future C++ Standard. More Boost libraries are proposed for the upcoming TR2".

Boost thrived and became a significant source of libraries and ideas for the standards committee and the C++ community in general.

5. Language Features: 1991-1998

By 1991, the most significant C++ language features for C++98 had been accepted: templates and exceptions as specified in the ARM were officially part of the language. However, work on their detailed specification went on for another several years. In addition, the committee worked on many new features, such as

1992 Covariant return types — the first extension beyond the features described in the ARM

1993 Run-time type identification (RTTI: `dynamic_cast`, `typeid`, and `type_info`); §5.1.2

1993 Declarations in conditions; §5.1.3

1993 Overloading based on enumerations

1993 Namespaces; §5.1.1

1993 `mutable`

1993 New casts (`static_cast`, `reinterpret_cast`, and `const_cast`)

1993 A Boolean type (`bool`); §5.1.4

1993 Explicit template instantiation

1993 Explicit template argument specification in function template calls

1994 Member templates ("nested templates")

1994 Class templates as template arguments

1996 In-class member initializers

1996 Separate compilation of templates (`export`); §5.2

1996 Template partial specialization

1996 Partial ordering of overloaded function templates

I won't go into detail here; the history of these features can be found in D&E [121] and TC++PL3 [126] describes their use. Obviously, most of these features were proposed and discussed long before they were voted into the standard.

Did the committee have overall criteria for acceptance of new features? Not really. The introduction of classes, class hierarchies, templates, and exceptions each (and in combination) represented a deliberate attempt to change the way people think about programming and write code. Such a major change was part of my aims for C++. However, as far as a committee can be said to think, that doesn't seem to be the way it does it. Individuals bring forward proposals, and the ones that make progress through the committee and reach a vote tend to be of limited scope. The committee members are busy and primarily practical people with little patience for abstract goals and a liking of concrete details that are amenable to exhaustive examination.

It is my opinion that the sum of the facilities added gives a more complete and effective support of the programming styles supported by C++, so we could say that the overall aim of these proposals is to "provide better support for procedural, object-oriented, and generic programming and for data abstraction". That's true, but it is not a concrete criterion that can be used to select proposals to work on from a long list. To the extent that the process has been successful in selecting new "minor features", it has been the result of decisions by individuals on a proposal-by-proposal basis. That's not my ideal, but the result could have been much worse. ISO C++ (C++98) is a better approximation to my ideals than the previous versions of C++ were. C++98 is a far more flexible (powerful) programming language than "ARM C++" (§3). The main reason for that is the cumulative effect of the refinements, such as member templates.

Not every feature accepted is in my opinion an improvement, though. For example, "in-class initialization of static const members of integral type with a constant expression" (proposed by John "Max" Skaller representing Australia and New Zealand) and the rule that `void f(T)` and `void f(const T)` denote the same function (proposed by Tom Plum for C compatibility reasons) share the dubious distinction of having been voted into C++ "over my dead body".

5.1 Some "Minor Features"

The "minor features" didn't feel minor when the committee worked on them, and may very well not look minor to a programmer using them. For example, I refer to namespaces and RTTI as "major" in D&E [121]. However, they don't significantly change the way we think about programs, so I will only briefly discuss a few of the features that many would deem "not minor".

5.1.1 Namespaces

C provides a single global namespace for all names that don't conveniently fit into a single function, a single `struct`, or a single translation unit. This causes problems with name clashes. I first grappled with this problem in the original design of C++ by defaulting all names to be local to a translation unit and requiring an explicit `extern` declaration to make them visible to other translation units. This idea was neither sufficient to solve the problem nor sufficiently compatible to be acceptable, so it failed.

When I devised the type-safe linkage mechanism [121], I reconsidered the problem. I observed that a slight change to the

```
extern "C" { /* ... */ }
```

syntax, semantics, and implementation technique would allow us to have

```
extern XXX { /* ... */ }
```

mean that names declared in `XXX` were in a separate scope `XXX` and accessible from other scopes only when qualified by `XXX::` in exactly the same way static class members are accessed from outside their class.

For various reasons, mostly related to lack of time, this idea lay dormant until it resurfaced in the ANSI/ISO committee discussions early in 1991. First, Keith Rowe from Microsoft presented a proposal that suggested the notation

```
bundle XXX { /* ... */ };
```

as a mechanism for defining a named scope and an operator `use` for bringing all names from a `bundle` into another scope. This led to a — not very vigorous — discussion among a few members of the extensions group including Steve Dovich, Dag Brück, Martin O'Riordan, and me. Eventually, Volker Bauche, Roland Hartinger, and Erwin Unruh from Siemens refined the ideas discussed into a proposal that didn't use new keywords:

```
:: XXX :: { /* ... */ };
```

This led to a serious discussion in the extensions group. In particular, Martin O'Riordan demonstrated that this `::` notation led to ambiguities with `::` used for class members and for global names.

By early 1993, I had — with the help of multi-megabyte email exchanges and discussions at the standards meetings — synthesized a coherent proposal. I recall technical contributions on namespaces from Dag Brück, John Bruns, Steve Dovich, Bill Gibbons, Philippe Gautron, Tony Hansen, Peter Juhl, Andrew Koenig, Eric Krohn, Doug McIlroy, Richard Minner, Martin O'Riordan, John "Max" Skaller, Jerry Schwarz, Mark Terribile, Mike Vilot, and me. In addition, Mike Vilot argued for immediate development of the ideas into a definite proposal so that the facilities would be available for addressing the inevitable naming problems in the ISO C++ library. In addition to various common C and C++ techniques for limiting the damage of name clashes, the facilities offered by Modula-2 and Ada were discussed. Namespaces were voted into C++ at the Munich meeting in July 1993. So, we can write:

```
namespace XXX {
   // ...
   int f(int);
}

int f(int);
int x = f(1);        // call global f
int y = XXX::f(1);   // call XXX's f
```

At the San Jose meeting in November 1993, it was decided to use namespaces to control names in the standard C and C++ libraries.

The original `namespace` design included a few more facilities, such as namespace aliases to allow abbreviations for long names, `using` declarations to bring individual names into a namespace, `using` directives to make all names from a namespace available with a single directive. Three years later, argument-dependent lookup (ADL or "Koenig lookup") was added to make namespaces of argument type names implicit.

The result was a facility that is useful and used but rarely loved. Namespaces do what they are supposed to do, sometimes elegantly, sometimes clumsily, and sometimes they do more than some people would prefer (especially argument-dependent lookup during template instantiation). The fact that the C++ standard library uses only a single namespace for all of its major facilities is an indication of a failure to establish namespaces as a primary tool of C++ programmers. Using sub-namespaces for the standard library would have implied a standardization of parts of the library implementation (to say which facilities were in which namespaces and which parts depended on other parts). Some library vendors strongly objected to such constraints on their traditional freedom as implementers — traditionally the internal organization of C and C++ libraries have been essentially unconstrained. Using sub-namespaces would also have been a source of verbosity. Argument-dependent lookup would have helped, but it was only introduced later in the standardization process. Also, ADL suffers from a bad interaction with templates that in some cases make it prefer a surprising template over an obvious non-template. Here "surprising" and "obvious" are polite renderings of user comments.

This has led to proposals for C++0x to strengthen namespaces, to restrict their use, and most interestingly a proposal from David Vandevoorde from EDG to make some namespaces into modules [146] — that is, to provide separately compiled namespaces that load as modules. Obviously, that facility looks a bit like the equivalent features of Java and C#.

5.1.2 Run-time type information

When designing C++, I had left out facilities for determining the type of an object (Simula's `QUA` and `INSPECT` similar to Smalltalk's `isKindOf` and `isA`). The reason was that I had observed frequent and serious misuse to the great detriment of program organization: people were using these facilities to implement (slow and ugly) versions of a switch statement.

The original impetus for adding facilities for determining the type of an object at run time to C++ came from Dmitry Lenkov from Hewlett-Packard. Dmitry in turn built on experience from major C++ libraries such as Interviews [81], the NIH library [50], and ET++ [152]. The RTTI mechanisms provided by these libraries (and others) were mutually incompatible, so they became a barrier to the use of more than one library. Also, all require considerable foresight from base class designers. Consequently, a language-supported mechanism was needed.

I got involved in the detailed design for such mechanisms as the coauthor with Dmitry of the original proposal to the committee and as the main person responsible for the refinement of the proposal in the committee [119]. The proposal was first presented to the committee at the London meeting in July 1991 and accepted at the Portland, Oregon meeting in March 1993.

The run-time type information mechanism consists of three parts:

- An operator, `dynamic_cast`, for obtaining a pointer to an object of a derived class given a pointer to a base class of that object. The operator `dynamic_cast` delivers that pointer only if the object pointed to really is of the specified derived class; otherwise it returns `0`.
- An operator, `typeid`, for identifying the exact type of an object given a pointer to a base class.
- A structure, `type_info`, acting as a hook for further run-time information associated with a type.

Assume that a library supplies class `dialog_box` and that its interfaces are expressed in terms of `dialog_box`es. I, however, use both `dialog_box`es and my own `Sbox`s:

```
class dialog_box : public window {
   // library class known to ``the system''
public:
   virtual int ask();
   // ...
};

class Sbox : public dialog_box {
   // can be used to communicate a string
public:
   int ask();
   virtual char* get_string();
   // ...
};
```

So, when the system/library hands me a pointer to a `dialog_box`, how can I know whether it is one of my `Sbox`s? Note that I can't modify the library to know my `Sbox` class. Even if I could, I wouldn't, because then I would have to modify every new version of the library forever after. So, when the system passes an object to my code, I sometimes need to ask it if was "one of mine". This question can be asked directly using the `dynamic_cast` operator:

```
void my_fct(dialog_box* bp)
{
   if (Sbox* sbp = dynamic_cast<Sbox*>(bp)) {
      // use sbp
   }
   else {
      // treat *pb as a ``plain'' dialog box
   }
}
```

The `dynamic_cast<T*>(p)` converts `p` to the desired type `T*` if `*p` really is a `T` or a class derived from `T`; otherwise, the value of `dynamic_cast<T*>(p)` is `0`. This use of `dynamic_cast` is the essential operation of a GUI callback. Thus, C++'s RTTI can be seen as the minimal facility for supporting a GUI.

If you don't want to test explicitly, you can use references instead of pointers:

```
void my_fct(dialog_box& br)
{
   Sbox& sbr = dynamic_cast<Sbox&>(br);
     // use sbr
}
```

Now, if the `dialog_box` isn't of the expected type, an exception is thrown. Error handling can then be elsewhere (§5.3).

Obviously, this run-time type information is minimal. This has led to requests for the maximal facility: a full meta-data facility (reflection). So far, this has been deemed unsuitable for a programming language that among other things is supposed to leave its applications with a minimal memory footprint.

5.1.3 Declarations in conditions

Note the way the cast, the declaration, and the test were combined in the "box example":

```
if (Sbox* sbp = dynamic_cast<Sbox*>(bp)) {
   // use sbp
}
```

This makes a neat logical entity that minimizes the chance of forgetting to test, minimizes the chance of forgetting to initialize, and limits the scope of the variable to its minimum. For example, the scope of `dbp` is the `if`-statement.

The facility is called "declaration in condition" and mirrors the "declaration as `for`-statement initializer", and "declaration as statement". The whole idea of allowing declarations everywhere was inspired by the elegant statements-as-expressions definition of Algol68 [154]. I was therefore most amazed when Charles Lindsey explained to me at the HOPL-II conference that Algol68 for technical reasons had not allowed declarations in conditions.

5.1.4 Booleans

Some extensions really are minor, but the discussions about them in the C++ community are not. Consider one of the most common enumerations:

```
enum bool { false, true };
```

Every major program has that one or one of its cousins:

```
#define bool char
#define Bool int
typedef unsigned int BOOL;
typedef enum { F, T } Boolean;
const true = 1;
#define TRUE 1
#define False (!True)
```

The variations are apparently endless. Worse, most variations imply slight variations in semantics, and most clash with other variations when used together.

Naturally, this problem has been well known for years. Dag Brück (representing Ericsson and Sweden) and Andrew Koenig (AT&T) decided to do something about it: "The idea of a Boolean data type in C++ is a religious issue. Some people, particularly those coming from Pascal or Algol, consider it absurd that C should lack such a type, let alone C++. Others, particularly those coming from C, consider it absurd that anyone would bother to add such a type to C++"

Naturally, the first idea was to define an `enum`. However, Dag Brück and Sean Corfield (UK) examined hundreds of thousands of lines of C++ and found that most Boolean types were used in ways that required implicit conversion of `bool` to and from `int`. C++ does not provide implicit conversion of `int`s to enumerations, so defining a standard `bool` as an enumeration would break too much existing code. So why bother with a Boolean type?

- The Boolean data type is a fact of life whether it is a part of a C++ standard or not.
- The many clashing definitions make it hard to use *any* Boolean type conveniently and safely.
- Many people want to overload based on a Boolean type.

Somewhat to my surprise, the committee accepted this argument, so `bool` is now a distinct integral type in C++ with literals `true` and `false`. Non-zero values can be implicitly converted to `true`, and `true` can be implicitly converted to `1`. Zero can be implicitly converted to `false`, and `false` can be implicitly converted to `0`. This ensures a high degree of compatibility.

Over the years, `bool` proved popular. Unexpectedly, I found it useful in teaching C++ to people without previous programming experience. After `bool`'s success in C++, the C standards committee decided to also add it to C. Unfortunately, they decided to do so in a different and incompatible way, so in C99 [64], `bool` is a `macro` for the keyword `_Bool` defined in the header `<stdbool.h>` together with macros `true` and `false`.

5.2 The Export Controversy

From the earliest designs, templates were intended to allow a template to be used after specifying just a declaration in a translation unit [117, 35]. For example:

```
template<class In, class T>
In find(In, In, const T&);   // no function body

vector<int>::iterator p =
   find(vi.begin(), vi.end(),42);
```

It is then the job of the compiler and linker to find and use the definition of the `find` template (D&E §15.10.4). That's the way it is for other language constructs, such as functions, but for templates that's easily said but extremely hard to do.

The first implementation of templates, Cfront 3.0 (October 1991), implemented this, but in a way that was very expensive in both compile time and link time. However, when Taumetric and Borland implemented templates, they introduced the "include everything" model: Just place all template definitions in header files and the compiler plus linker will eliminate the multiple definitions you get when you include a file multiple times in separately compiled translation units. The First Borland compiler with "rudimentary template support" shipped November 20, 1991, quickly followed by version 3.1 and the much more robust version 4.0 in November 1993 [27]. Microsoft, Sun, and others followed along with (mutually incompatible) variations of the "include everything" approach. Obviously, this approach violates the usual separation between an implementation (using definitions) and an interface (presenting only declarations) and makes definitions vulnerable to macros, unintentional overload resolution, etc. Consider a slightly contrived example:

```
// printer.h:
   template<class Destination>
   class Printer {
      locale loc;
      Destination des;
   public:
      template<class T> void out(const T& x)
         { print(des,x,loc); }
```

```
    // ...
  };
```

We might use `Printer` in two translation units like this:

```
//user1.c:
  typedef int locale; // represent locale by int
  #define print(a,b,c) a(c)<<x
  #include "printer.h"
  // ...
```

and this

```
//user2.c:
  #include<locale>  // use standard locale
  using namespace std;
  #include "printer.h"
  // ...
```

This is obviously illegal because differences in the contexts in which `printer.h` are seen in `user1.c` and `user2.c` lead to inconsistent definitions of `Printer`. However, such errors are hard for a compiler to detect. Unfortunately, the probability of this kind of error is far higher in C++ than in C and even higher in C++ using templates than in C++ that doesn't use templates. The reason is that when we use templates there is so much more text in header files for typedefs, overloads, and macros to interfere with. This leads to defensive programming practices (such as naming all local names in template definitions in a cryptic style unlikely to clash, e.g., `_L2`) and a desire to reduce the amount of code in the header files through separate compilation.

In 1996, a vigorous debate erupted in the committee over whether we should not just accept the "include everything" model for template definitions, but actually outlaw the original model of separation of template declarations and definitions into separate translation units. The arguments of the two sides were basically

- Separate translation of templates is too hard (if not impossible) and such a burden should not be imposed on implementers
- Separate translation of templates is necessary for proper code organization (according to data-hiding principles)

Many subsidiary arguments supported both sides. Mike Ball (Sun) and John Spicer (EDG) led the "ban separate compilation of templates" group and Dag Bruck (Ericcson and Sweden) usually spoke for the "preserve separate compilation of templates" group. I was on the side that insisted on separate compilation of templates. As ever in really nasty discussions, both sides were mostly correct on their key points. In the end, people from SGI — notably John Wilkinson — proposed a new model that was accepted as a compromise. The compromise was named after the keyword used to indicate that a template could be separately translated: `export`.

The separate compilation of templates issue festers to this day: The "export" feature remains disabled even in some compilers that do support it because enabling it would break ABIs. As late as 2003, Herb Sutter (representing Microsoft) and Tom Plum (of Plum Hall) proposed a change to the standard so that an implementation that didn't implement separate compilation of templates would still be conforming; that is, `export` would be an optional language feature. The reason given was again implementation complexity plus the fact that even five years after the standard was ratified only one implementation existed. That motion was defeated by an 80% majority, partly because an implementation of `export` now existed. Independently of the technical arguments, many committee members considered it unfair to deem a feature optional after some, but not all, implementers had spent significant time and effort implementing it.

The real heroes of this sad tale are the implementers of the EDG compiler: Steve Adamczyk, John Spicer, and David Vandevoorde. They strongly opposed separate compilation of templates, finally voted for the standard as the best compromise attainable, and then proceeded to spend more than a year implementing what they had opposed. That's professionalism! The implementation was every bit as difficult as its opponents had predicted, but it worked and provided some (though not all) of the benefits that its proponents had promised. Unfortunately, some of the restrictions on separately compiled templates that proved essential for the compromise ended up not providing their expected benefits and complicated the implementation. As ever, political compromises on technical issues led to "warts".

I suspect that one major component of a better solution to the separate compilation of templates is concepts (§8.3.3) and another is David Vandevoorde's modules [146].

5.3 Exception Safety

During the effort to specify the STL we encountered a curious phenomenon: We didn't quite know how to talk about the interaction between templates and exceptions. Quite a few people were placing blame for this problem on templates and others began to consider exceptions fundamentally flawed (e.g., [20]) or at least fundamentally flawed in the absence of automatic garbage collection. However, when a group of "library people" (notably Nathan Myers, Greg Colvin, and Dave Abrahams) looked into this problem, they found that we basically had a language feature — exceptions — that we didn't know how to use well. The problem was in the interaction between resources and exceptions. If throwing an exception renders resources inaccessible there is no hope of recovering gracefully. I had of course considered this when I designed the exception-handling mechanisms and come up with the rules for exceptions thrown from constructors (correctly handling partially constructed composite objects) and the "resource acquisition is initialization" technique (§5.3.1). However, that was only a good start and an essential foundation. What we needed was a conceptual framework — a more systematic way of thinking about resource management. Together with many other people, notably Matt Austern, such a framework was developed.

Dave Abrahams condensed the result of work over a couple of years into three guarantees [1]:

- The *basic guarantee*: that the invariants of the component are preserved, and no resources are leaked.
- The *strong guarantee*: that the operation has either completed successfully or thrown an exception, leaving the program state exactly as it was before the operation started.
- The *no-throw guarantee*: that the operation will not throw an exception.

Note that the strong guarantee basically is the database "commit or rollback" rule. Using these fundamental concepts, the library working group described the standard library and implementers produced efficient and robust implementations. The standard library provides the basic guarantee for all operations with the caveat that we may not exit a destructor by throwing an exception. In addition, the library provides the strong guarantee and the no-throw guarantee for key operations. I found this result important enough to add an appendix to TC++PL [124], yielding [126]. For details of the standard library exception guarantees and programming techniques for using exceptions, see Appendix E of TC++PL.

The first implementations of the STL using these concepts to achieve exception safety were Matt Austern's SGI STL and Boris Fomitch's STLPort [42] augmented with Dave Abrahams' exception-safe implementations of standard containers and algorithms. They appeared in the spring of 1997.

I think the key lesson here is that it is not sufficient just to know how a language feature behaves. To write good software, we must have a clearly articulated design strategy for problems that require the use of the feature.

5.3.1 Resource management

Exceptions are typically — and correctly — seen as a control structure: a `throw` transfers control to some `catch`-clause. However, sequencing of operations is only part of the picture: error handling using exceptions is mostly about resource management and invariants. This view is actually built into the C++ class and exception primitives in a way that provides a necessary foundation for the guarantees and the standard library design.

When an exception is thrown, every constructed object in the path from the `throw` to the `catch` is destroyed. The destructors for partially constructed objects (and unconstructed objects) are not invoked. Without those two rules, exception handling would be unmanageable (in the absence of other support). I (clumsily) named the basic technique "resource acquisition is initialization" — commonly abbreviated to "RAII". The classical example [118] is

```
// naive and unsafe code:
void use_file(const char* fn)
{
   FILE* f = fopen(fn,"w");  // open file fn
   // use f
   fclose(f);  // close file fn
}
```

This looks plausible. However, if something goes wrong after the call of `fopen` and before the call of `fclose`, an exception may cause `use_file` to be exited without calling `fclose`. Please note that exactly the same problem can occur in languages that do not support exception handling. For example, a call of the standard C library function `longjmp` would have the same bad effects. Even a misguided `return` among the code using `f` would cause the program to leak a file handle. If we want to support writing fault-tolerant systems, we must solve this problem.

The general solution is to represent a resource (here the file handle) as an object of some class. The class' constructor acquires the resource and the class' destructor gives it back. For example, we can define a class `File_ptr` that acts like a `FILE*`:

```
class File_ptr {
   FILE* p;
public:
   File_ptr(const char* n, const char* a)
   {
      p = fopen(n,a);
      if (p==0) throw Failed_to_open(n);
   }
   ~File_ptr() { fclose(p); }
   // copy, etc.
   operator FILE*() { return p; }
};
```

We can construct a `File_ptr` given the arguments required for `fopen`. The `File_ptr` will be destroyed at the end of its scope and its destructor closes the file. Our program now shrinks to this minimum

```
void use_file(const char* fn)
{
   File_ptr f(fn,"r");  // open file fn
   // use f
} // file fn implicitly closed
```

The destructor will be called independently of whether the function is exited normally or because an exception is thrown.

The general form of the problem looks like this:

```
void use()
{
   // acquire resource 1
   // ...
   // acquire resource n

   // use resources

   // release resource n
   // ...
```

```
      // release resource 1
   }
```

This applies to any "resource" where a resource is anything you acquire and have to release (hand back) for a system to work correctly. Examples include files, locks, memory, iostream states, and network connections. Please note that automatic garbage collection is *not* a substitute for this: the point in time of the release of a resource is often important logically and/or performance-wise.

This is a systematic approach to resource management with the important property that correct code is shorter and less complex than faulty and primitive approaches. The programmer does not have to remember to do anything to release a resource. This contrasts with the older and ever-popular `finally` approach where a programmer provides a `try`-block with code to release the resource. The C++ variant of that solution looks like this:

```
   void use_file(const char* fn)
   {
      FILE* f = fopen(fn,"r");  // open file fn
      try {
         // use f
      }
      catch (...) {   // catch all
         fclose(f);   // close file fn
         throw;       // re-throw
      }
      fclose(f);      // close file fn
}
```

The problem with this solution is that it is verbose, tedious, and potentially expensive. It can be made less verbose and tedious by providing a `finally` clause in languages such as Java, C#, and earlier languages [92]. However, shortening such code doesn't address the fundamental problem that the programmer has to remember to write release code for each acquisition of a resource rather just once for each resource, as in the RAII approach. The introduction of exceptions into the ARM and their presentation as a conference paper [79] was delayed for about half a year until I found "resource acquisition is initialization" as a systematic and less error-prone alternative to the `finally` approach.

5.4 Automatic Garbage Collection

Sometime in 1995, it dawned on me that a majority of the committee was of the opinion that plugging a garbage collector into a C++ program was not standard-conforming because the collector would inevitably perform some action that violated a standard rule. Worse, they were obviously right about the broken rules. For example:

```
   void f()
   {
      int* p = new int[100];
      // fill *p with valuable data
      file << p;  // write the pointer to a file
      p = 0;      // remove the pointer to the ints
      // work on something else for a week
      file >> p;
      if (p[37] == 5) {        // now use the ints
        // ...
      }
   }
```

My opinion — as expressed orally and in print — was roughly: "such programs deserve to be broken" and "it is perfectly good C++ to use a conservative garbage collector". However, that wasn't what the draft standard said. A garbage collector would undoubtedly have recycled that memory before we read the pointer back from the file and started using the integer array again. However, in standard C and standard C++, there is absolutely nothing that allows a piece of memory to be recycled without some explicit programmer action.

To fix this problem, I made a proposal to explicitly allow "optional automatic garbage collection" [123]. This would bring C++ back to what I had thought I had defined it to be and make the garbage collectors already in actual use [11, 47] standard conforming. Explicitly mentioning this in the standard would also encourage use of GC where appropriate. Unfortunately, I seriously underestimated the dislike of garbage collection in a large section of the committee and also mishandled the proposal.

My fatal mistake with the GC proposal was to get thoroughly confused about the meaning of "optional". Did "optional" mean that an implementation didn't have to provide a garbage collector? Did it mean that the programmer could decide whether the garbage collector was turned on or not? Was the choice made at compile time or run time? What should happen if I required the garbage collector to be activated and the implementation didn't supply one? Can I ask if the garbage collector is running? How? How can I make sure that the garbage collector isn't running during a critical operation? By the time a confused discussion of such questions had broken out and different people had found conflicting answers attractive, the proposal was effectively dead.

Realistically, garbage collection wouldn't have passed in 1995, even if I hadn't gotten confused. Parts of the committee

- strongly distrusted GC for performance reasons
- disliked GC because it was seen as a C incompatibility
- didn't feel they understood the implications of accepting GC (we didn't)
- didn't want to build a garbage collector
- didn't want to pay for a garbage collector (in terms of money, space, or time)
- wanted alternative styles of GC
- didn't want to spend precious committee time on GC

Basically, it was too late in the standards process to introduce something that major. To get anything involving garbage collection accepted, I should have started a year earlier.

My proposal for garbage collection reflected the then major use of garbage collection in C++ — that is, conservative collectors that don't make assumptions about which memory locations contain pointers and never move objects around in memory [11]. Alternative approaches included creating a type-safe subset of C++ so that it is possible to know exactly where every pointer is, using smart pointers [34] and providing a separate operator (`gcnew` or `new(gc)`) for allocating objects on a "garbage-collected heap". All three approaches are feasible, provide distinct benefits, and have proponents. This further complicates any effort to standardize garbage collection for C++.

A common question over the years has been: Why don't you add GC to C++? Often, the implication (or follow-up comment) is that the C++ committee must be a bunch of ignorant dinosaurs not to have done so already. First, I observe that in my considered opinion, C++ would have been stillborn had it relied on garbage collection when it was first designed. The overheads of garbage collection at the time, on the hardware available, precluded the use of garbage collection in the hardware-near and performance-critical areas that were C++'s bread and butter. There were garbage-collected languages then, such as Lisp and Smalltalk, and people were reasonably happy with those for the applications for which they were suitable. It was not my aim to replace those languages in their established application areas. The aim of C++ was to make object-oriented and data-abstraction techniques affordable in areas where these techniques at the time were "known" to be impractical. The core areas of C++ usage involved tasks, such as device drivers, high-performance computation, and hard-real-time tasks, where garbage collection was (and is) either infeasible or not of much use.

Once C++ was established without garbage collection and with a set of language features that made garbage collection difficult (pointers, casts, unions, etc.), it was hard to retrofit it without doing major damage. Also, C++ provides features that make garbage collection unnecessary in many areas (scoped objects, destructors, facilities for defining containers and smart pointers, etc.). That makes the case for garbage collection less compelling.

So, why would I like to see garbage collection supported in C++? The practical reason is that many people write software that uses the free store in an undisciplined manner. In a program with hundreds of thousands of lines of code with `new`s and `delete`s all over the place, I see no hope for avoiding memory leaks and access through invalid pointers. My main advice to people who are starting a project is simply: "don't do that!". It is fairly easy to write correct and efficient C++ code that avoids those problems through the use of containers (STL or others; §4.1), resource handles (§5.3.1, and (if needed) smart pointers (§6.2). However, many of us have to deal with older code that does deal with memory in an undisciplined way, and for such code plugging in a conservative garbage collector is often the best option. I expect C++0x to require every C++ implementation to be shipped with a garbage collector that, somehow, can be either active or not.

The other reason that I suspect a garbage collector will eventually become necessary is that I don't see how to achieve perfect type safety without one — at least not without pervasive testing of pointer validity or damaging compatibility (e.g. by using two-word non-local pointers). And improving type safety (e.g., "eliminate every implicit type violation" [121]) has always been a fundamental long-term aim of C++. Obviously, this is a very different argument from the usual "when you have a garbage collector, programming is easy because you don't have to think about deallocation". To contrast, my view can be summarized as "C++ is such a good garbage-collected language because it creates so little garbage that needs to be collected". Much of the thinking about C++ has been focused on resources in general (such as locks, file handles, thread handles, and free store memory). As noted in §5.3, this focus has left traces in the language itself, in the standard library, and in programming techniques. For large systems, for embedded systems, and for safety-critical systems a systematic treatment of resources seems to me much more promising than a focus on garbage collection.

5.5 What Wasn't Done

Choosing what to work on is probably more significant than how that work is done. If someone — in this case the C++ standards committee — decides to work on the wrong problem, the quality of the work done is largely irrelevant. Given the limited time and resources, the committee could cope with only a few topics and choose to work hard on those few rather than take on more topics. By and large, I think that the committee chose well and the proof of that is that C++98 is a significantly better language than ARM C++. Naturally, we could have done better still, but even in retrospect it is hard to know how. The decisions made at the time were taken with as much information (about problems, resources, possible solutions, commercial realities, etc.) as we had available then and who are we to second guess today?

Some questions are obvious, though:

- Why didn't we add support for concurrency?
- Why didn't we provide a much more useful library?
- Why didn't we provide a GUI?

All three questions were seriously considered and in the first two cases settled by explicit vote. The votes were close to unanimous. Given that we had decided not to pursue concurrency or to provide a significantly larger library, the question about a GUI library was moot.

Many of us — probably most of the committee members — would have liked some sort of concurrency support. Concurrency is fundamental and important. Among other suggestions, we even had a pretty solid proposal for concurrency support in the form of micro-C++ from the University of Toronto [18]. Some of us, notably Dag Brück relying on data from Ericssson, looked at the issue and presented the case for *not* dealing with concurrency in the committee:

- We found the set of alternative ways of supporting concurrency large and bewildering.
- Different application areas apparently had different needs and definitely had different traditions.
- The experience with Ada's direct language support for concurrency was discouraging.
- Much (though of course not all) could be done with libraries.
- The committee lacked sufficient experience to do a solid design.
- We didn't want a set of primitives that favored one approach to concurrency over others.
- We estimated that the work — if feasible at all — would take years given our scarce resources.
- We wouldn't be able to decide what other ideas for improvements to drop to make room for the concurrency work.

My estimate at the time was that "concurrency is as big a topic as all of the other extensions we are considering put together". I do not recall hearing what I in retrospect think would have been "the killer argument": Any sufficient concurrency support will involve the operating system; since C++ is a systems programming language, we need to be able to map the C++ concurrency facilities to the primitives offered. But for each platform the owners insist that C++ programmers can use every facility offered. In addition, as a fundamental part of a multi-language environment, C++ cannot rely on a concurrency model that is dramatically different from what other languages support. The resulting design problem is so constrained that it has no solution.

The reason the lack of concurrency support didn't hurt the C++ community more than it did is that much of what people actually do with concurrency is pretty mundane and can be done through library support and/or minor (non-standard) language extensions. The various threads libraries (§8.6) and MPI [94] [49] offer examples.

Today, the tradeoffs appear to be different: The continuing increase in gate counts paired with the lack of increase of hardware clock speeds is a strong incentive to exploit low-level concurrency. In addition, the growth of multiprocessors and clusters requires other styles of concurrency support and the growth of wide-area networking and the web makes yet other styles of concurrent systems essential. The challenge of supporting concurrency is more urgent than ever and the major players in the C++ world seem far more open to the need for changes to accommodate it. The work on C++0x reflects that (§8.2).

The answer to "Why didn't we provide a much more useful library?" is simpler: We didn't have the resources (time and people) to do significantly more than we did. Even the STL caused a year's delay and gaining consensus on other components, such as `iostreams`, strained the committee. The obvious shortcut — adopting a commercial foundation library — was considered. In 1992, Texas Instruments offered their very nice library for consideration and within an hour five representatives of major corporations made it perfectly clear that if this offer was seriously considered they would propose their own corporate foundation libraries. This wasn't a way that the committee could go. Another committee with a less consensus-based culture might have made progress by choosing one commercial library over half-a-dozen others in major use, but not this C++ committee.

It should also be remembered that in 1994, many already considered the C++ standard library monstrously large. Java had not yet changed programmers' perspective of what they could expect "for free" from a language. Instead, many in the C++ community used the tiny C standard library as the measure of size. Some national bodies (notably the Netherlands and France) repeatedly expressed worry that C++ standard library was seriously bloated. Like many in the committee, I also hoped that the standard would help rather than try to supplant the C++ libraries industry.

Given those general concerns about libraries, the answer to "Why didn't we provide a GUI?" is obvious: The committee couldn't do it. Even before tackling the GUI-specific design issues, the committee would have had to tackle concurrency and settle on a container design. In addition, many of the members were simply blindsided. GUI was seen as just another large and complex library that people — given `dynamic_cast` (§5.1.2) — could write themselves (in particular, that was my view). They did. The problem today is not that there is no C++ GUI library, but that there are on the order of 25 such libraries in use (e.g., Gtkmm [161], FLTK [156], SmartWin++ [159], MFC [158], WTL [160], vxWidgets (formerly wxWindows) [162], Qt [10]). The committee could have worked on a GUI library, worked on library facilities that could be used as a basis for GUI libraries, or worked on standard library interfaces to common GUI functionality. The latter two approaches might have yielded important results, but those paths weren't taken and I don't think that the committee then had the talents necessary for success in that direction. Instead, the committee worked hard on more elaborate interfaces to stream I/O. That was probably a dead end because the facilities for multiple character sets and locale dependencies were not primarily useful in the context of traditional data streams.

6. Standards Maintenance: 1997-2005

After a standard is passed, the ISO process can go into a "maintenance mode" for up to five years. The C++ committee decided to do that because:

- the members were tired (after ten years' work for some members) and wanted to do something else,
- the community was way behind in understanding the new features,
- many implemeters were way behind with the new features, libraries, and support tools,
- no great new ideas were creating a feeling of urgency among the members or in the community,
- for many members, resources (time and money) for standardization were low, and
- many members (including me) thought that the ISO process required a "cooling-off period".

In "maintenance mode," the committee primarily responded to defect reports. Most defects were resolved by clarifying the text or resolving contradictions. Only very rarely were new rules introduced and real innovation was avoided. Stability was the aim. In 2003, all these minor corrections were published under the name "Technical Corrigendum 1". At the same time, members of the British national committee took the opportunity to remedy a long-standing problem: they convinced Wiley to publish a printed version of the (revised) standard [66]. The initiative and much of the hard work came from Francis Glassborow and Lois Goldthwaite with technical support from the committee's project editor, Andrew Koenig, who produced the actual text.

Until the publication of the revised standard in 2003, the only copies of the standard available to the public were a very expensive (about $200) paper copy from ISO or a cheap ($18) pdf version from INCITS (formerly ANSI X3). The pdf version was a complete novelty at the time. Standards bodies are partially financed through the sales of standards, so they are most reluctant to make them available free of charge or cheaply. In addition, they don't have retail sales channels, so you can't find a national or international standard in your local book store — except the C++ standard, of course. Following the C++ initiative, the C standard is now also available.

Much of the best standards work is invisible to the average programmer and appears quite esoteric and often boring when presented. The reason is that a lot of effort is expended in finding ways of expressing clearly and completely "what everyone already knows, but just happens not to be spelled out in the manual" and in resolving obscure issues that — at least in theory — don't affect most programmers. The maintenance is mostly such "boring and esoteric" issues. Furthermore, the committee necessarily focuses on issues where the standard contradicts itself — or appears to do so. However, these issues are essential to implementers trying to ensure that a given language use is correctly handled. In turn, these issues become essential to programmers because even the most carefully written large program will deliberately or accidentally depend on some feature that would appear obscure or esoteric to some. Unless implementers agree, the programmer has a hard time achieving portability and easily becomes the hostage of a single compiler purveyor — and that would be contrary to my view of what C++ is supposed to be.

To give an idea of the magnitude of this maintenance task (which carries on indefinitely): Since the 1998 standard until 2006, the core and library working groups has each handled on the order of 600 "defect reports". Fortunately, not all were real defects, but even determining that there really is no problem, or that the problem is just lack of clarity in the standard's text, takes time and care.

Maintenance wasn't all that the committee did from 1997 to 2003. There was a modest amount of planning for the future (thinking about C++0x), but the main activities were writing a technical report on performance issues [67] and one on libraries [68].

6.1 The Performance TR

The performance technical report ("TR") [67] was prompted by a suggestion to standardize a subset of C++ for embedded systems programming. The proposal, called Embedded C++ [40] or simply EC++, originated from a consortium of Japanese embedded systems tool developers (including Toshiba, Hitachi, Fujitsu, and NEC) and had two main concerns: removal of language features that potentially hurt performance and removal of language features perceived to be too complicated for programmers (and thus seen as potential productivity or correctness hazards). A less clearly stated aim was to define something that in the short term was easier to implement than full standard C++.

The features banned in this (almost) subset included: multiple inheritance, templates, exceptions, run-time type information (§5.1.2), new-style casts, and name spaces. From the standard library, the STL and locales were banned and an alternative version of `iostreams` provided. I considered the proposal misguided and backwards looking. In particular, the performance costs were largely imaginary, or worse. For example, the use of templates has repeatedly been shown to be key to both performance (time and space) and correctness of embedded systems. However, there wasn't much hard data in this area in 1996 when EC++ was first proposed. Ironically, it appears that most of the few people who use EC++ today use it in the form of "Extended EC++" [36], which is EC++ plus templates. Similarly, namespaces (§5.1.1) and new style casts (§5) are features that are primarily there to clarify code and can be used to ease maintenance and verification of correctness. The best documented (and most frequently quoted) overhead of "full C++" as compared to EC++ was `iostreams`. The primary reason for that is that the C++98 `iostreams` support locales whereas that older `iostreams` do not. This is somewhat ironic because the locales were added to support languages different from English (most notably Japanese) and can be optimized away in environments where they are not used (see [67]).

After serious consideration and discussion, the ISO committee decided to stick to the long-standing tradition of not endorsing dialects — even dialects that are (almost) subsets. Every dialect leads to a split in the user community, and so does even a formally defined subset when its users start to develop a separate culture of techniques, libraries, and tools. Inevitably, myths about failings of the full language relative to the "subset" will start to emerge. Thus, I recommend against the use of EC++ in favor of using what is appropriate from (full) ISO Standard C++.

Obviously, the people who proposed EC++ were right in wanting an efficient, well-implemented, and relatively easy-to-use language. It was up to the committee to demonstrate that ISO Standard C++ was that language. In particular, it seemed a proper task for the committee to document the utility of the features rejected by EC++ in the context of performance-critical, resource-constrained, or safety-critical tasks. It was therefore decided to write a technical report on "performance". Its executive summary reads:

> 'The aim of this report is:
>
> - to give the reader a model of time and space overheads implied by use of various C++ language and library features,
> - to debunk widespread myths about performance problems,
> - to present techniques for use of C++ in applications where performance matters, and
> - to present techniques for implementing C++ Standard language and library facilities to yield efficient code.
>
> As far as run-time and space performance is concerned, if you can afford to use C for an application, you can afford to use C++ in a style that uses C++'s facilities appropriately for that application.

Not every feature of C++ is efficient and predictable to the extent that we need for some high-performance and embedded applications. A feature is predictable if we can in advance easily and precisely determine the time needed for each use. In the context of an embedded system, we must consider if we can use

- free store (`new` and `delete`)
- run-time type identification (`dynamic_cast` and `typeid`)
- exceptions (throw and catch)

The time needed to perform one of these operations can depend on the context of the code (e.g. how much stack unwinding a `throw` must perform to reach its matching `catch`) or on the state of the program (e.g. the sequence of `new`s and `delete`s before a `new`).

Implementations aimed at embedded or high performance applications all have compiler options for disabling run-time type identification and exceptions. Free store usage is easily avoided. All other C++ language features are predictable and can be implemented optimally (according to the zero-overhead principle; §2). Even exceptions can be (and tend to be) efficient compared to alternatives [93] and should be considered for all but the most stringent hard-real-time systems. The TR discusses these issues and also defines an interface to the lowest accessible levels of hardware (such as registers). The performance TR was written by a working group primarily consisting of people who cared about embedded systems, including members of the EC++ technical committee. I was active in the performance working group and drafted significant portions of the TR, but the chairman and editor was first Martin O'Riordan and later Lois Goldthwaite. The acknowledgments list 28 people, including Jan Kristofferson, Dietmar Kühl, Tom Plum, and Detlef Vollmann. In 2004, that TR was approved by unanimous vote.

In 2004, after the TR had been finalized, Mike Gibbs from Lockheed-Martin Aero found an algorithm that allows `dynamic_cast` to be implemented in constant time, and fast [48]. This offers hope that `dynamic_cast` will eventually be usable for hard-real-time programming.

The performance TR is one of the few places where the immense amount of C++ usage in embedded systems surfaces in publicly accessible writing. This usage ranges from high-end systems such as found in telecommunications systems to really low-level systems where complete and direct access to specific hardware features is essential (§7). To serve the latter, the performance TR contains a "hardware addressing interface" together with guidelines for its usage. This interface is primarily the work of Jan Kristofferson (representing Ramtex International and Denmark) and Detlef Vollmann (representing Vollmann Engineering GmbH and Switzerland). To give a flavor, here is code copying a register buffer specified by a port called `PortA2_T`:

```
unsigned char mybuf[10];
register_buffer<PortA2_T, Platform> p2;
for (int i = 0; i != 10; ++i)
{
   mybuf[i] = p2[i];
}
```

Essentially the same operation can be done as a block read:

```
register_access<PortA3_T, Platform> p3;
UCharBuf myBlock;
myBlock = p3;
```

Note the use of templates and the use of integers as template arguments; it's essential for a library that needs to maintain optimal performance in time and space. This comes as a surprise to many who have been regaled with stories of memory bloat caused by templates. Templates are usually implemented by generating a copy of the code used for each specialization; that is, for each combination of template arguments. Thus, obviously, if your code generated by a template takes up a lot of memory, you can use a lot of memory.

However, many modern templates are written based on the observation that an inline function may shrink to something that's as small as the function preamble or smaller still. That way you can simultaneously save both time and space. In addition to good inlining, there are two more bases for good performance from template code: dead code elimination and avoidance of spurious pointers.

The standard requires that no code is generated for a member function of a class template unless that member function is called for a specific set of template arguments. This automatically eliminates what could have become "dead code". For example:

```
template<class T> class X {
public:
      void f() { /* ... */ }
      void g() { /* ... */ }
      // ...
};

int main()
{
      X<int> xi;
      xi.f();
      X<double> xd;
      xd.g();
      return 0;
 }
```

For this program, the compiler must generate code for `X<int>::f()` and `X<double>::g()` but may not generate code for `X<int>::g()` and `X<double>::f()`. This rule was added to the standard in 1993, at my insistence, specifically to reduce code bloat, as observed in early uses of templates. I have seen this used in embedded systems in the form of the rule "all classes must be templates, even if only a single instantiation is used". That way, dead code is automatically eliminated.

The other simple rule to follow to get good memory performance from templates is: "don't use pointers if you don't need them". This rule preserves complete type information and allows the optimizer to perform really well (especially when inline functions are used). This implies that function templates that take simple objects (such as function objects) as arguments should do so by value rather than by reference. Note that pointers to functions and `virtual` functions break this rule, causing problems for optimizers.

It follows that to get massive code bloat, say megabytes, what you need is to

1. use large function templates (so that the code generated is large)
2. use lots of pointers to objects, virtual functions, and pointers to functions (to neuter the optimizer)
3. use "feature rich" hierarchies (to generate a lot of potentially dead code)
4. use a poor compiler and a poor optimizer

I designed templates specifically to make it easy for the programmer to avoid (1) and (2). Based on experience, the standard deals with (3), except when you violate (1) or (2). In the early 1990s, (4) became a problem. Alex Stepanov named it "the abstraction penalty" problem. He defined "the abstraction penalty" as the ratio of runtime between a templated operation (say, `find` on a `vector<int>` and the trivial non-templated equivalent (say a loop over an array of `int`). An implementation that does all of the easy and obvious optimizations gets a ratio of 1. Poor compilers had an abstraction penalty of 3, though even then good implementations did significantly better. In October 1995, to encourage implementers to do better, Alex wrote the "abstraction penalty benchmark", which simply measured the abstraction penalty [102]. Compiler and optimizer writers didn't like their implementations to be obviously poor, so today ratios of 1.02 or so are common.

The other — and equally important — aspect of C++'s support for embedded systems programming is simply that its model of computation and memory is that of real-world hardware: the built-in types map directly to memory and registers, the built-in operations map directly to machine operations, and the composition mechanisms for data structures do not impose spurious indirections or memory overhead [131]. In this, C++ equals C. See also §2.

The views of C++ as a close-to-machine language with abstraction facilities that can be used to express predicable type-safe low-level facilities has been turned into a coding standard for safety-critical hard-real-time code by Lockheed-Martin Aero [157]. I helped draft that standard. Generally, C and assembly language programmers understand the direct mapping of language facilities to hardware, but often not the the need for abstraction mechanisms and strong type checking. Conversely, programmers brought up with higher-level "object-oriented" languages often fail to see the need for closeness to hardware and expect some unspecified technology to deliver their desired abstractions without unacceptable overheads.

6.2 The Library TR

When we finished the standard in 1997, we were fully aware that the set of standard libraries was simply the set that we had considered the most urgently needed and also ready to ship. Several much-wanted libraries, such as hash tables, regular expression matching, directory manipulation, and threads, were missing. Work on such libraries started immediately in the Libraries Working Group chaired by Matt Austern (originally working at SGI with Alex Stepanov, then at AT&T Labs with me, and currently at Google). In 2001, the committee started work on a technical report on libraries. In 2004 that TR [68] specifying libraries that people considered most urgently needed was approved by unanimous vote.

Despite the immense importance of the standard library and its extensions, I will only briefly list the new libraries here:

- Polymorphic function object wrapper
- Tuple types
- Mathematical special functions
- Type traits
- Regular expressions
- Enhanced member pointer adaptor
- General-purpose smart pointers
- Extensible random number facility
- Reference wrapper
- Uniform method for computing function-object return types
- Enhanced binder
- Hash tables

Prototypes or industrial-strength implementations of each of these existed at the time of the vote; they are expected to ship with every new C++ implementation from 2006 onwards. Many of these new library facilities are obviously "technical"; that is, they exist primarily to support library builders. In particular, they exist to support builders of standard library facilities in the tradition of the STL. Here, I will just emphasize three libraries that are of direct interest to large numbers of application builders:

- Regular expressions
- General-purpose smart pointers
- Hash tables

Regular expression matching is one of the backbones of scripting languages and of much text processing. Finally, C++ has a standard library for that. The central class is `regex`, providing regular expression matching of patterns compatible with ECMAscript (formerly JavaScript or Jscript) and with other popular notations.

The main "smart pointer" is a reference-counted pointer, `shared_ptr`, intended for code where shared ownership is needed. When the last `shared_ptr` to an object is destroyed, the object pointed to is deleted. Smart pointers are popular, but not universally so and concerns about their performance and likely overuse kept `smart_ptr`'s "ancestor", `counted_ptr`, out of C++98. Smart pointers are not the panacea they are sometimes presented to be. In particular, they can be far more expensive to use than ordinary pointers, destructors for objects "owned" by a set of `shared_ptrs` will run at unpredictable times, and if a lot of objects are deleted at once because the last `shared_ptr` to them is deleted you can incur "garbage-collection delays" exactly as if you were running a general collector. The costs primarily relate to free-store allocation of use-count objects and especially to locking during access to the use counts in threaded systems ("lock-free" implementations appear to help here). If it is garbage collection you want, you might be better off simply using one of the available garbage collectors [11, 47] or waiting for C++0x (§5.4).

No such worries affected hash tables; they would have been in C++98 had we had the time to do a proper detailed design and specification job. There was no doubt that a `hash_map` was needed as an alternative to `map` for large tables where the key was a character string and we could design a good hash function. In 1995, Javier Barreirro, Robert Fraley and David Musser tried to get a proposal ready in time for the standard and their work became the basis for many of the later `hash_maps` [8]. The committee didn't have the time, though, and consequently the Library TR's `unordered_map` (and `unordered_set`) are the result of about eight years of experiment and industrial use. A new name, "unordered_map", was chosen because now there are half a dozen incompatible `hash_maps` in use. The `unordered_map` is the result of a consensus among the `hash_map` implementers and their key users in the committee. An `unordered_map` is "unordered" in the sense that an iteration over its elements are not guaranteed to be in any particular order: a hash function doesn't define an ordering in the way a `map`'s < does.

The most common reaction to these extensions among developers is "that was about time; why did it take you so long?" and "I want much more right now". That's understandable (I too want much more right now — I just know that I can't get it), but such statements reflect a lack of understanding what an ISO committee is and can do. The committee is run by volunteers and requires both a consensus and an unusual degree of precision in our specifications (see D&E §6.2). The committee doesn't have the millions of dollars that commercial vendors can and do spend on "free", "standard" libraries for their customers.

7. C++ in Real-World Use

Discussions about programming languages typically focus on language features: which does the language have? how efficient are they? More enlightened discussions focus on more difficult questions: how is the language used? how can it be used? who can use it? For example, in an early OOPSLA keynote, Kristen Nygaard (of Simula and OOP fame) observed: "if we build languages that require a PhD from MIT to use, we have failed". In industrial contexts, the first — and often only — questions are: Who uses the language? What for? Who supports it? What are the alternatives? This section presents C++ and its history from the perspective of its use.

Where is C++ used? Given the number of users (§1), it is obviously used in a huge number of places, but since most of the use is commercial it is difficult to document. This is one of the many areas where the lack of a central organization for C++ hurts the C++ community — nobody systematically gathers information about its use and nobody has anywhere near complete information. To give an idea of the range of application areas, here are a few examples where C++ is used for crucial components of major systems and applications:

- Adobe — Acrobat, Photoshop, Illustrator, ...
- Amadeus — airline reservations
- Amazon — e-commerce
- Apple — iPod interface, applications, device drivers, finder, ...
- AT&T — 1-800 service, provisioning, recovery after network failure, ...
- eBay — online auctions
- Games — Doom3, StarCraft, Halo, ...
- Google — search engines, Google Earth, ...
- IBM — K42 (very high-end operating system), AS/400, ...
- Intel — chip design and manufacturing, ...
- KLA-Tencor — semiconductor manufacturing
- Lockheed-Martin Aero — airplane control (F16, JSF), ...
- Maeslant Barrier — Dutch surge barrier control
- MAN B&W — marine diesel engine monitoring and fuel injection control
- Maya — professional 3D animation
- Microsoft — Windows XP, Office, Internet explorer, Visual Studio, .Net, C# compiler, SQL, Money, ...
- Mozilla Firefox — browser
- NASA/JPL — Mars Rover scene analysis and autonomous driving, ...
- Southwest Airlines — customer web site, flight reservations, flight status, frequent flyer program, ...
- Sun — compilers, OpenOffice, HotSpot Java Virtual Machine, ...
- Symbian — OS for hand-held devices (especially cellular phones)
- Vodaphone — mobile phone infrastructure (including billing and provisioning)

For more examples, see [137]. Some of the most widely used and most profitable software products ever are on this list. Whatever C++'s theoretical importance and impact, it certainly met its most critical design aim: it became an immensely useful practical tool. It brought object-oriented programming and more recently also generic programming into the mainstream. In terms of numbers of applications and the range of application areas, C++ is used beyond any individual's range of expertise. That vindicates my emphasis on generality in D&E [121] (Chapter 4).

It is a most unfortunate fact that applications are not documented in a way that reaches the consciousness of researchers, teachers, students, and other application builders. There is a huge amount of experience "out there" that isn't documented or made accessible. This inevitably warps people's sense of reality — typically in the direction of what is new (or perceived as new) and described in trade press articles, academic papers, and textbooks. Much of the visible information is very open to fads and commercial manipulation. This leads to much "reinvention of the wheel", suboptimal practice, and myths.

One common myth is that "most C++ code is just C code compiled with a C++ compiler". There is nothing wrong with such code — after all, the C++ compiler will find more bugs than a C compiler — and such code is not uncommon. However, from seeing a lot of commercial C++ code and talking with innumerable developers and from talking with developers, I know that for many major applications, such as the ones mentioned here, the use of C++ is far more "adventurous". It is common for developers to mention use of major locally designed class hierarchies, STL use, and use of "ideas from STL" in local code. See also §7.2.

Can we classify the application areas in which C++ is used? Here is one way of looking at it:

- Applications with systems components
- Banking and financial (funds transfer, financial modeling, customer interaction, teller machines, ...)
- Classical systems programming (compilers, operating systems, editors, database systems, ...)
- Conventional small business applications (inventory systems, customer service, ...)
- Embedded systems (instruments, cameras, cell phones, disc controllers, airplanes, rice cookers, medical systems, ...)
- Games
- GUI — iPod, CDE desktop, KDE desktop, Windows, ...
- Graphics
- Hardware design and verification [87]
- Low-level system components (device drivers, network layers, ...)
- Scientific and numeric computation (physics, engineering, simulations, data analysis, ...)
- Servers (web servers, large application backbones, billing systems, ...)
- Symbolic manipulation (geometric modeling, vision, speech recognition, ...)
- Telecommunication systems (phones, networking, monitoring, billing, operations systems, ...)

Again, this is more a list than a classification. The world of programming resists useful classification. However, looking at these lists, one thing should be obvious: C++ cannot be ideal for all of that. In fact, from the earliest days of C++, I have maintained that a general-purpose language can at most be second best in a well-defined application area and that C++ is "a general-purpose programming language with a bias towards systems programming".

7.1 Applications Programming vs. Systems Programming

Consider an apparent paradox: C++ has a bias towards systems programming but "most programmers" write applications. Obviously millions of programmers are writing applications and many of those write their applications in C++. Why? Why don't they write them in an applications programming language? For some definition of "applications programming language", many do. We might define an "applications programming language" as one in which we can't directly access hardware and that provides direct and specialized support for an important applications concept (such as data base access, user interaction, or numerical computation). Then, we can deem most languages "applications languages": Excel, SQL, RPG, COBOL, Fortran, Java, C#, Perl, Python, etc. For good and bad, C++ is used as a general-purpose programming language in many areas where a more specialized (safer, easier to use, easier to optimize, etc.) language would seem applicable.

The reason is not just inertia or ignorance. I don't claim that C++ is anywhere near perfect (that would be absurd) nor that it can't be improved (we are working on C++0x, after all and see §9.4). However, C++ does have a niche — a very large niche — where other current languages fall short:

- applications with a significant systems programming component; often with resource constraints
- applications with components that fall into different application areas so that no single specialized applications language could support all

Application languages gain their advantages through specialization, through added conveniences, and through eliminating difficult to use or potentially dangerous features. Often, there is a run-time or space cost. Often, simplifications are based on strong assumptions about the execution environment. If you happen to need something fundamental that was deemed unnecessary in the design (such as direct access to memory or fully general abstraction mechanisms) or don't need the "added conveniences" (and can't afford the overhead they impose), C++ becomes a candidate. The basic conjecture on which C++ is built is that many applications have components for which that is the case: "Most of us do unusual things some of the time".

The positive way of stating this is that general mechanisms beat special-purpose features for the majority of applications that does not completely fit into a particular classification, have to collaborate with other applications, or significantly evolve from their original narrow niche. This is one reason that every language seems to grow general-purpose features, whatever its original aims were and whatever its stated philosophy is.

If it was easy and cheap to switch back and forth among applications languages and general-purpose languages, we'd have more of a choice. However, that is rarely the case, especially where performance or machine-level access is needed. In particular, using C++ you can (but don't have to) break some fundamental assumption on which an application language is built. The practical result is that if you need a systems programming or performance-critical facility of C++ somewhere in an application, it becomes convenient to use C++ for a large part of the application — and then C++'s higher-level (abstraction) facilities come to the rescue. C++ provides hardly any high-level features that are directly applicable in an application. What it offers are mechanisms for defining such facilities as libraries.

Please note that from a historical point of view this analysis need not be correct or the only possible explanation of the facts. Many prefer alternative ways of looking at the problem. Successful languages and companies have been built on alternative views. However, it is a fact that C++ was designed based on this view and that this view guided the evolution of C++; for example, see Chapter 9 of [121]. I consider it the reason that C++ initially succeeded in the mainstream and the reason that its use continued to grow steadily during the time period covered by this paper, despite the continuing presence of well-designed and better-financed alternatives in the marketplace. See also §9.4.

7.2 Programming Styles

C++ supports several programming styles or, as they are sometimes somewhat pretentiously called, "programming paradigms". Despite that, C++ is often referred to as "an object-oriented programming language". This is only true for some really warped definition of "object-oriented" and I never say just "C++ is an object-oriented language" [122]. Instead, I prefer "C++ supports object-oriented programming and other programming styles" or "C++ is a multi-paradigm programming language". Programming style matters. Consequently, the way people refer to a language matters because it sets expectations and influences what people see as ideals.

C++ has C (C89) as an "almost subset" and supports the styles of programming commonly used for C [127]. Arguably, C++ supports those styles better than C does by providing more type checking and more notational support. Consequently, a lot of C++ code has been written in the style of C or — more commonly — in the style of C with a number of classes and class hierarchies thrown in without affecting the overall design. Such code is basically procedural, using classes to provide a richer set of types. That's sometimes referred to as "C with Classes style". That style can be significantly better (for understanding, debugging, maintenance, etc.) than pure C. However, it is less interesting from a historical perspective than code that also uses the C++ facilities to express more advanced programming techniques, and very often less effective than such alternatives. The fraction of C++ code written purely in "C style" appears to have been decreasing over the last 15 years (or more).

The abstract data type and object-oriented styles of C++ usage have been discussed often enough not to require explanation here (e.g., see [126]). They are the backbone of many C++ applications. However, there are limits to their utility. For example, object-oriented programming can lead to overly rigid hierarchies and overreliance on virtual functions. Also, a virtual function call is fundamentally efficient for cases where you need to select an action at run time, but it is still an indirect function call and thus expensive compared to an individual machine instruction. This has led to generic programming becoming the norm for C++ where high performance is essential (§6.1).

7.2.1 Generic programming

Sometimes, generic programming in C++ is defined as simply "using templates". That's at best an oversimplification. A better description from a programming language feature point of view is "parametric polymorphism" [107] plus overloading, which is selecting actions and constructing types based on parameters. A template is basically a compile-time mechanism for generating definitions (of classes and functions) based on type arguments, integer arguments, etc. [144].

Before templates, generic programming in C++ was done using macros [108], `void*`, and casts, or abstract classes. Naturally, some of that still persists in current use and occasionally these techniques have advantages (especially when combined with templatized interfaces). However, the current dominant form of generic programming relies on class templates for defining types and function templates for defining operations (algorithms).

Being based on parameterization, generic programming is inherently more regular than object-oriented programming. One major conclusion from the years of use of major generic libraries, such as the STL, is that the current support for generic programming in C++ is insufficient. C++0x is taking care of at least part of that problem (§8).

Following Stepanov (§4.1.8), we can define generic programming without mentioning language features: Lift algorithms and data structures from concrete examples to their most general and abstract form. This typically implies representing the algorithms and their access to data as templates, as shown in the descripton of the STL (§4.1).

7.2.2 Template metaprogramming

The C++ template instantiation mechanism is (when compiler limits are ignored, as they usually can be) Turing complete (e.g., see [150]). In the design of the template mechanism, I had aimed at full generality and flexibility. That generality was dramatically illustrated by Erwin Unruh in the early days of the standardization of templates. At an extensions working group meeting in 1994, he presented a program that calculated prime numbers at compile time (using error messages as the output mechanism) [143] and appeared surprised that I (and others) thought that marvelous rather than scary. Template instantiation is actually a small compile-time functional programming language. As early as 1995, Todd Veldhuizen showed how to define a compile-time if-statement using templates and how to use such if-statements (and switch-statements) to select among alternative data structures and algorithms [148]. Here is a compile-time if-statement with a simple use:

```
template<bool b, class X, class Y>
struct if_ {
   typedef X type;  // use X if b is true
};

template<class X, class Y>
struct if_<false,X,Y> {
     typedef Y type;  // use Y if b is false
};

void f()
{
   if_<sizeof(Foobar)<40, Foo, Bar>::type xy;
   // ...
}
```

If the size of type `Foobar` is less than `40`, the type of the variable `xy` is `Foo`; otherwise it is `Bar`. The second definition of `if_` is a partial specialization used when the template arguments match the `<false,X,Y>` pattern specified. It's really quite simple, but very ingenious and I remember being amazed when Jeremy Siek first showed it to me. In a variety of guises, it has proven useful for producing portable high-performance libraries (e.g., most of the Boost libraries [16] rely on it).

Todd Veldhuizen also contributed the technique of expression templates [147], initially as part of the implementation of his high-performance numeric library Blitz++ [149]. The key idea is to achieve compile-time resolution and delayed evaluation by having an operator return a function object containing the arguments and operation to be (eventually) evaluated. The < operator generating a `Less_than` object in §4.1.4 is a trivial example. David Vandevoorde independently discovered this technique.

These techniques and others that exploit the computational power of template instantiation provide the foundation for techniques based on the idea of generating source code that exactly matches the needs of a given situation. It can lead to horrors of obscurity and long compile times, but also to elegant and very efficient solutions to hard problems; see [3, 2, 31]. Basically, template instantiation relies on overloading and specialization to provide a reasonably complete functional compile-time programming language.

There is no universally agreed-upon definition of the distinction between generic programming and template metaprogramming. However, generic programming tends to emphasize that each template argument type must have an enumerated well-specified set of properties; that is, it must be able to define a concept for each argument (§4.1.8, §8.3.3). Template metaprogramming doesn't always do that. For example, as in the `if_` example, template definitions can be chosen based on very limited aspects of an argument type, such as its size. Thus, the two styles of programming are not completely distinct. Template metaprogramming blends into generic programming as more and more requirements are placed on arguments. Often, the two styles are used in combination. For example, template metaprogramming can be used to select template definitions used in a generic part of a program.

When the focus of template use is very strongly on composition and selection among alternatives, the style of programming is sometimes called "generative programming" [31].

7.2.3 Multi-paradigm programming

It is important for programmers that the various programming styles supported by C++ are part of a single language. Often, the best code requires the use of more than one of the four basic "paradigms". For example, we can write the classical "draw all shapes in a container" example from SIMULA BEGIN [9] like this:

```
void draw_all(vector<Shape*>& v)
{
   for_each(v.begin(), v.end(),     // sequence
            mem_fun(&Shape::draw)); // operation
}
```

Here, we use object-oriented programming to get the runtime polymorphism from the `Shape` class hierarchy. We use generic programming for the parameterized (standard library) container `vector` and the parameterized (standard library) algorithm `for_each`. We use ordinary procedural programming for the two functions `draw_all` and `mem_fun`. Finally, the result of the call of `mem_fun` is a function object, a class that is not part of a hierarchy and has no virtual functions, so that can be classified as abstract data type programming. Note that `vector`, `for_each`, `begin`, `end`, and `mem_fun` are templates, each of which will generate the most appropriate definition for its actual use.

We can generalize that to any sequence defined by a pair of `ForwardIterator`s, rather than just `vector`s and improve type checking using C++0x concepts (§8.3.3):

```
template<ForwardIterator For>
void draw_all(For first, For last)
   requires SameType<For::value_type,Shape*>
{
   for_each(first, last, mem_fun(&Shape::draw));
}
```

I consider it a challenge to properly characterize multi-paradigm programming so that it can be easy enough to use for most mainstream programmers. This will involve finding a more descriptive name for it. Maybe it could even benefit from added language support, but that would be a task for C++1x.

7.3 Libraries, Toolkits, and Frameworks

So, what do we do when we hit an area in which the C++ language is obviously inadequate? The standard answer is: Build a library that supports the application concepts. C++ isn't an application language; it is a language with facilities supporting the design and implementation of elegant and efficient libraries. Much of the talk about object-oriented programming and generic programming comes down to building and using libraries. In addition to the standard library (§4) and the components from the library TR (§6.2), examples of widely used C++ libraries include

- ACE [95] — distributed computing
- Anti-Grain Geometry — 2D graphics
- Borland Builder (GUI builder)
- Blitz++[149] — vectors "The library that thinks it is a compiler"
- Boost[16] — foundation libraries building on the STL, graph algorithms, regular expression matching, threading, ...
- CGAL[23] — computational geometry
- Maya — 3D animation
- MacApp and PowerPlant — Apple foundation frameworks
- MFC — Microsoft Windows foundation framework
- Money++ — banking
- RogueWave library (pre-STL foundation library)
- STAPL[4], POOMA[86] — parallel computation
- Qt [10], FLTK [156], gtkmm [161], wxWigets [162] — GUI libraries and builders
- TAO [95], MICO, omniORB — CORBA ORBs
- VTK [155] — visualization

In this context, we use the word "toolkit" to describe a library supported by programming tools. In this sense, VTK is a toolkit because it contains tools for generating interfaces in Java and Python and Qt and FLTK are toolkits because they provide GUI builders. The combination of libraries and tools is an important alternative to dialects and special-purpose languages [133, 151].

A library (toolkit, framework) can support a huge user community. Such user communities can be larger than the user communities of many programming languages and a library can completely dominate the world-view of their users. For example, Qt [10] is a commercial product with about 7,500 paying customers in 2006 plus about 150,000 users of its open-source version [141]. Two Norwegian programmers, Eirik Chambe-Eng and Haavard Nord, started what became Qt in 1991-92 and the first commercial release was in 1995. It is the basis of the popular desktop KDE (for Linux, Solaris, and FreBSD) and well known commercial products, such as Adobe Photoshop Elements, Google Earth, and Skype (Voice over IP service).

Unfortunately for C++'s reputation, a good library cannot be seen; it just does its job invisibly to its users. This often leads people to underestimate the use of C++. However, "there are of course the Windows Foundation Classes (MFC), MacApp, and PowerPlant — most Mac and Windows commercial software is built with one of these frameworks" [85].

In addition to these general and domain-specific libraries, there are many much more specialized libraries. These have their function limited to a specific organization or application. That is, they apply libraries as an application design philosophy: "first extend the language by a library, then write the application in the resulting extended language". The "string library" (part of a larger system called "Panther") used by Celera Genomics as the base for their work to sequence the human genome [70] is a spectacular example of this approach. Panther is just one of many C++ libraries and applications in the general area of biology and biological engineering.

7.4 ABIs and Environments

Using libraries on a large scale isn't without problems. C++ supports source-level compatibility (but provides only weak link-time compatibility guarantees). That's fine if

- you have (all) the source
- your code compiles with your compiler
- the various parts of your source code are compatible (e.g., with respect to resource usage and error handling)
- your code is all in C++

For a large system, typically none of these conditions hold. In other words, linking is a can of worms. The root of this problem is the fundamental C++ design decision: Use existing linkers (D&E §4.5).

It is not guaranteed that two C translation units that match according to the language definition will link correctly when compiled with different compilers. However, for every platform, agreement has been reached for an ABI (Application Binary Interface) so that the register usage, calling conventions, and object layout match for all compilers so that C programs will correctly link. C++ compilers use these conventions for function call and simple structure layout. However, traditionally C++ compiler vendors have resisted linkage standards for layout of class hierarchies, virtual function calls, and standard library components. The result is that to be sure that a legal C++ program actually works, every part (including all libraries) has to be compiled by the same compiler. On some platforms, notably Sun's and also Itanium (IA64) [60], C++ ABI standards exist but historically the rule "use a single compiler or communicate exclusively through C functions and `structs`" is the only really viable rule.

Sticking with one compiler can ensure link compatibility on a platform, but it can also be a valuable tool in providing portability across many platforms. By sticking to a single implementer, you gain "bug compatibility" and can target all platforms supported by that vendor. For Microsoft platforms, Microsoft C++ provides that opportunity; for a huge range of platforms, GNU C++ is portable; and for a diverse set of platforms, users get a pleasant surprise when they notice that many of their local implementations use an EDG (Edison Design Group) front-end making their source code portable. This only (sic!) leaves the problems of version skew. During this time period every C++ compiler went through a series of upgrades, partly to increase standard conformance, to adjust to platform ABIs, and to improve performance, debugging, integration with IDEs, etc.

Link compatibility with C caused a lot of problems, but also yielded significant advantages. Sean Parent (Adobe) observes: "one reason I see for C++'s success is that it is 'close enough' to C that platform vendors are able to provide a single C interface (such as the Win32 API or the Mac Carbon API) which is C++ compatible. Many libraries provide a C interface which is C++ compatible because the effort to do so is low — where providing an interface to a language such as Eiffel or Java would be a significant effort. This goes beyond just keeping the linking model the same as C but to the actual language compatibility".

Obviously, people have tried many solutions to the linker problem. The platform ABIs are one solution. CORBA is a platform- and language-independent (or almost so) solution that has found widespread use. However, it seems that C++ and Java are the only languages heavily used with CORBA. COM was Microsoft's platform-dependent and language-independent solution (or almost so). One of the origins of Java was a perceived need to gain platform independence and compiler independence; the JVM solved that problem by eliminating language independence and completely specifying linkage. The Microsoft CLI (Common Language Infrastructure) solves the problem in a language-independent manner (sort of) by requiring all languages to support a Java-like linkage, metadata, and execution model. Basically all of these solutions provide platform independence by becoming a platform: To use a new machine or operating system, you port a JVM, an ORB, etc.

The C++ standard doesn't directly address the problem of platform incompatibilities. Using C++, platform independence is provided through platform-specific code (typically relying on conditional compilation — `#ifdef`). This is often messy and ad hoc, but a high degree of platform independence can be provided by localizing dependencies in the implementation of a relatively simple platform layer — maybe just a single header file [16]. In any case, to implement the platform-independent services effectively, you need a language that can take advantage of the peculiarities of the various hardware and operating systems environments. More often than not, that language is C++.

Java, C#, and many other languages rely on metadata (that is, data that defines types and services associated with them) and provide services that depend on such metadata (such as marshalling of objects for transfer to other computers). Again, C++ takes a minimalist view. The only "metadata" available is the RTTI (§5.1.2), which provides just the name of the class and the list of its base classes. When RTTI was discussed, some of us dreamed of tools that would provide more data for systems that needed it, but such tools did not become common, general-purpose, or standard.

7.5 Tools and Research

Since the late 1980s, C++ developers have been supported by a host of analysis tools, development tools, and development environments available in the C++ world. Examples are:

- Visual Studio (IDE; Microsoft)
- KDE (Desktop Environment; Free Software)
- Xcode (IDE; Apple)
- lint++ (static analysis tool; Gimpel Software)
- Vtune (multi-level performance tuning; Intel)
- Shark (performance optimization; Apple)
- PreFAST (static source code analysis; Microsoft)
- Klocwork (static code analysis; Klocwork)
- LDRA (testing; LDRA Ltd.)
- QA.C++ (static analysis; Programming Research)
- Purify (memory leak finder; IBM Rational)
- Great Circle (garbage collector and memory usage analyzer; Geodesic, later Symantec)

However, tools and environments have always been a relative weakness of C++. The root of that problem is the difficulty of parsing C++. The grammar is not LR(N) for any N. That's obviously absurd. The problem arose because C++ was based directly on C (I borrowed a YACC-based C parser from Steve Johnson), which was "known" not to be expressible as a LR(1) grammar until Tom Pennello discovered how to write one in 1985. Unfortunately, by then I had defined C++ in such a way that Pennello's techniques could not be applied to C++ and there was already too much code dependent on the non-LR grammar to change C++. Another aspect of the parsing problem is the macros in the C preprocessor. They ensure that what the programmer sees when looking at a line of code can — and often does — dramatically differ from what the compiler sees when parsing and type checking that same line of code. Finally, the builder of advanced tools must also face the complexities of name lookup, template instantiation, and overload resolution.

In combination, these complexities have confounded many attempts to build tools and programming environments for C++. The result was that far too few software development and source code analysis tools have become widely used. Also, the tools that were built tended to be expensive. This led to relatively fewer academic experiments than are conducted with languages that were easier to analyze. Unfortunately, many take the attitude that "if it isn't standard, it doesn't exist" or alternatively "if it costs money, it doesn't exist". This has led to lack of knowledge of and underuse of existing tools, leading to much frustration and waste of time.

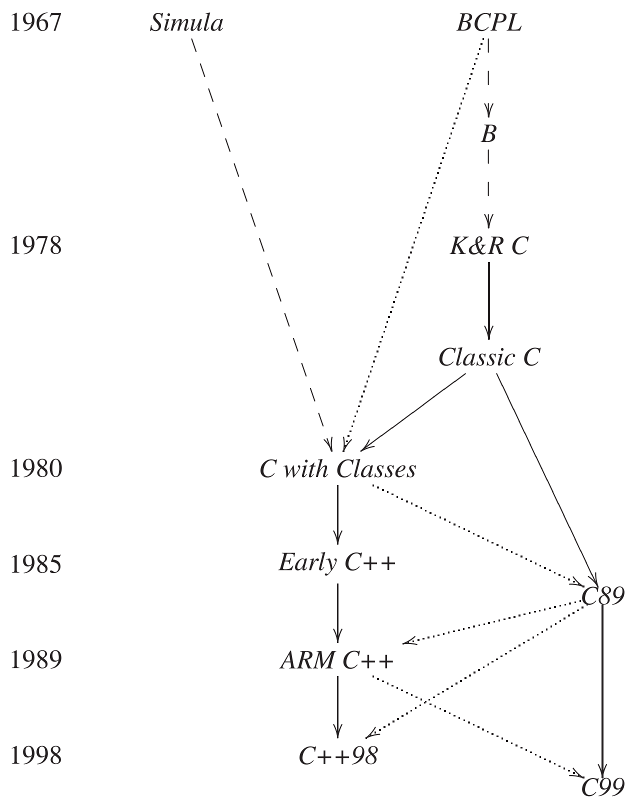
Indirectly, this parsing problem has caused weaknesses in areas that rely on run-time information, such as GUI-builders. Since the language doesn't require it, compilers generally don't produce any form of metadata (beyond the minimum required by RTTI; §5.1.2). My view (as stated in the original RTTI papers and in D&E) was that tools could be used to produce the type information needed by a specific application or application area. Unfortunately, the parsing problem then gets in the way. The tool-to-generate-metadata approach has been successfully used for database access systems and GUI, but cost and complexity have kept this approach from becoming more widely used. In particular, academic research again suffered because a typical student (or professor) doesn't have the time for serious infrastructure building.

A focus on performance also plays a part in lowering the number and range of tools. Like C, C++ is designed to ensure minimal run-time and space overheads. For example, the standard library `vector` is not by default range checked because you can build an optimal range-checked vector on top of an unchecked vector, but you cannot build an optimally fast vector on top of a range-checked one (at least not portably). However, many tools rely on additional actions (such as range-checking array access or validating pointers) or additional data (such as meta-data describing the layout of a data structure). A subculture of strong concern about performance came into the C++ community with much else from C. Often that has been a good thing, but it did have a limiting effect on tool building by emphasizing minimalism even where there were no serious performance issues. In particular, there is no reason why a C++ compiler couldn't supply superb type information to tool builders [134].

Finally, C++ was a victim of its own success. Researchers had to compete with corporations that (sometimes correctly) thought that there was money to be made in the kind of tools researchers would like to build. There was also a curious problem with performance: C++ was too efficient for any really significant gains to come easily from research. This led many researchers to migrate to languages with glaring inefficiencies for them to eliminate. Elimination of virtual function calls is an example: You can gain much better improvements for just about any object-oriented language than for C++. The reason is that C++ virtual function calls are very fast and that colloquial C++ already uses non-virtual functions for time-critical operations. Another example is garbage collection. Here the problem was that colloquial C++ programs don't generate much garbage and that the basic operations are fast. That makes the fixed overhead of a garbage collector looks far less impressive when expressed as a percentage of run time than it does for a language with less efficient basic operations and more garbage. Again, the net effect was to leave C++ poorer in terms of research and tools.

7.6 C/C++ Compatibility

C is C++'s closest relative and a high degree of C compatibility has always been a design aim for C++. In the early years, the primary reasons for compatibility were to share infrastructure and to guarantee completeness (§2.2). Later, compatibility remained important because of the huge overlap in applications areas and programmer communities. Many little changes were made to C++ during the '80s and '90s to bring C++ closer to ISO C [62] (C89). However, during 1995-2004, C also evolved. Unfortunately, C99 [64] is in significant ways less compatible with C++ than C89 [62] and harder to coexist with. See [127] for a detailed discussion of the C/C++ relationship. Here is a diagram of the relationships among the various generations of C and C++:



"Classic C" is what most people think of as K&R C, but C as defined in [76] lacks structure copy and enumerations. ARM C++ is C++ as defined by the ARM [35] and the basis for most pre-standard C++. The basic observation is that by now C (i.e., C89 or C99) and C++ (i.e., C++98) are siblings (with "Classic C" as their common ancestor), rather than the more conventional view that C++ is a dialect of C that somehow failed to be compatible. This is an important issue because people commonly proclaim the right of each language to evolve separately, yet just about everybody expects ISO C++ to adopt the features adopted by ISO C — despite the separate evolution of C and a tendency of the C committee to adopt features that are similar to but incompatible with what C++ already offers. Examples selected from a long list [127] are `bool`, `inline` functions, and `complex` numbers.

7.7 Java and Sun

I prefer not to compare C++ to other programming languages. For example, in the "Notes to the Reader" section of D&E [121], I write:

> Several reviewers asked me to compare C++ to other languages. This I have decided against doing. ... Language comparisons are rarely meaningful and even less often fair. A good comparison of major programming languages requires more effort than most people are willing to spend, experience in a wide range of application areas, a rigid maintenance of a detached and impartial point of view, and a sense of fairness. I do not have the time, and as the designer of C++, my

> impartiality would never be fully credible. ... Worse, when one language is significantly better known than others, a subtle shift in perspective occurs: Flaws in the well-known language are deemed minor and simple workarounds are presented, whereas similar flaws in other languages are deemed fundamental. Often, the workarounds commonly used in the less-well-known languages are simply unknown to the people doing the comparison or deemed unsatisfactory because they would be unworkable in the more familiar language. ... Thus, I restrict my comments about languages other than C++ to generalities and to very specific comments.

However, many claims about the C++/Java relationship have been made and the presence of Java in the marketplace has affected the C++ community. Consequently, a few comments are unavoidable even though a proper language comparison is far beyond the scope of this paper and even though Java has left no traces in the C++ definition.

Why not? From the earliest days of Java, the C++ committee has always included people with significant Java experience: users, implementers, tool builders, and JVM implementers. I think at a fundamental level Java and C++ are too different for easy transfer of ideas. In particular,

- C++ relies on direct access to hardware resources to achieve many of its goals whereas Java relies on a virtual machine to keep it away from the hardware.
- C++ is deliberately frugal with run-time support whereas Java relies on a significant amount of metadata
- C++ emphasizes interoperability with code written in other languages and sharing of system tools (such as linkers) whereas Java aims for simplicity by isolating Java code from other code.

The "genes" of C++ and Java are quite dissimilar. The syntactic similarities between Java and C++ have often been deceptive. As an analogy, I note that it is far easier for English to adopt "structural elements" from closely related languages, such as French or German, than from more different languages, such as Japanese or Thai.

Java burst onto the programming scene with an unprecedented amount of corporate backing and marketing (much aimed at non-programmers). According to key Sun people (such as Bill Joy), Java was an improved and simplified C++. "What Bjarne would have designed if he hadn't had to be compatible with C" was — and amazingly still is — a frequently heard statement. Java is not that; for example, in D&E §9.2.2, I outlined fundamental design criteria for C++:

> What would be a better language than C++ for the things C++ is meant for? Consider the first-order decisions:
>
> - Use of static type checking and Simula-like classes.
> - Clean separation between language and environment.
> - C source compatibility ("as close as possible").
> - C link and layout compatibility ("genuine local variables").
> - No reliance on garbage collection.
>
> I still consider static type checking essential for good design and run-time efficiency. Were I to design a new language for the kind of work done in C++ today, I would again follow the Simula model of type checking and inheritance, *not* the Smalltalk or Lisp models. As I have said many times, 'Had I wanted an imitation Smalltalk, I would have built a much better imitation. Smalltalk is the best Smalltalk around. If you want Smalltalk, use it'.

I think I could express that more clearly today, but the essence would be the same; these criteria are what define C++ as a systems programming language and what I would be unwilling to give up. In the light of Java, that section seems more relevant today than when I wrote it in 1993 (pre-Java). Having the built-in data types and operators mapped directly to hardware facilities and being able to exploit essentially every machine resource is implicit in "C compatibility".

C++ does not meet Java's design criteria; it wasn't meant to. Similarly, Java doesn't meet C++'s design criteria. For example, consider a couple of language-technical criteria:

- Provide as good support for user-defined types as for built-in types
- Leave no room for a lower-level language below C++ (except assembler)

Many of the differences can be ascribed to the aim of keeping C++ a systems programming language with the ability to deal with hardware and systems at the lowest level and with the least overhead. Java's stated aims seem more directed towards becoming an applications language (for some definition of "application").

Unfortunately, the Java proponents and their marketing machines did not limit themselves to praising the virtues of Java, but stooped to bogus comparisons (e.g., [51]) and to name calling of languages seen as competitors (most notably C and C++). As late as 2001, I heard Bill Joy claim (orally with a slide to back it up in a supposedly technical presentation) that "reliable code cannot be written in C/C++ because they don't have exceptions" (see §5, §5.3). I see Java as a Sun commercial weapon aimed at Microsoft that missed and hit an innocent bystander: the C++ community. It hurt many smaller language communities even more; consider Smalltalk, Lisp, Eiffel, etc.

Despite many promises, Java didn't replace C++ ("Java will completely kill C++ within two years" was a graphic expression I repeatedly heard in 1996). In fact, the C++ community has trebled in size since the first appearance of Java. The Java hype did, however, harm the C++ community by diverting energy and funding away from much-needed tools, library and techniques work. Another problem was that Java encouraged a limited "pure object-oriented" view of programming with a heavy emphasis on run-time resolution and a de-emphasis of the static type system (only in 2005 did Java introduce "generics"). This led many C++ programmers to write unnecessarily inelegant, unsafe, and poorly performing code in imitation. This problem gets especially serious when the limited view infects education and creates a fear of the unfamiliar in students.

As I predicted [136] when I first heard the boasts about Java's simplicity and performance, Java rapidly accreted new features — in some cases paralleling C++'s earlier growth. New languages are always claimed to be "simple" and to become useful in a wider range of real-world applications they increase in size and complexity. Neither Java nor C++ was (or is) immune to that effect. Obviously, Java has made great strides in performance — given its initial slowness it couldn't fail to — but the Java object model inhibits performance where abstraction is seriously used (§4.1.1, §4.1.4). Basically, C++ and Java are far more different in aims, language structure, and implementation model than most people seem to think. One way of viewing Java is as a somewhat restricted version of Smalltalk's run-time model hidden behind a C++-like syntax and statically type-checked interfaces.

My guess is that Java's real strength compared to C++ is the binary compatibility gained from Sun having de facto control of the linker as expressed through the definition of the Java virtual machine. This gives Java the link-compatibility that has eluded the C++ community because of the decision to use the basic system linkers and the lack of agreement among C++ vendors on key platforms (§7.4).

In the early 1990s, Sun developed a nice C++ compiler based on Mike Ball and Steve Clamage's Taumetric compiler. With the release of Java and Sun's loudly proclaimed pro-Java policy where C++ code was referred to (in advertising "literature" and elsewhere) as "legacy code" that needed rewriting and as "contamination", the C++ group suffered some lean years. However, Sun never wavered in its support of the C++ standards efforts and Mike, Steve and others made significant contributions. Technical people have to live with technical realities — such as the fact that many Sun customers and many Sun projects rely on C++ (in total or in part). In particular, Sun's Java implementation, HotSpot, is a C++ program.

7.8 Microsoft and .Net

Microsoft is currently the 800-pound gorilla of software development and it has had a somewhat varied relationship with C++. Their first attempt at an object-oriented C in the late 1980s wasn't C++ and I have the impression that a certain ambivalence about standard conformance lingers. Microsoft is better known for setting de facto standards than for strictly sticking to formal ones. However, they did produce a C++ compiler fairly early. Its main designer was Martin O'Riordan, who came from the Irish company Glockenspiel where he had been a Cfront expert and produced and maintained many ports. He once amused himself and his friends by producing a Cfront that spoke its error messages through a voice synthesizer in (what he believed to be) a thick Danish accent. To this day there are ex-Glockenspiel Irishmen on the Microsoft C++ team.

Unfortunately, that first release didn't support templates or exceptions. Eventually, those key features were supported and supported well, but that took years. The first Microsoft compiler to provide a close-to-complete set of ISO C++ features was VC++ 6.0, released in July 1998; its predecessor, VC++ 5.0 from February 1997, already had many of the key features. Before that, some Microsoft managers used highly visible platforms, such as conference keynotes, for some defensive bashing of these features (as provided only by their competitors, notably Borland) as "expensive and useless". Worse, internal Microsoft projects (which set widely followed standards) couldn't use templates and exceptions because their compiler didn't support those features. This established bad programming practices and did long-term harm.

Apart from that, Microsoft was a responsible and attentive member of the community. Microsoft sent and sends members to the committee meetings and by now — somewhat belatedly — provides an excellent C++ compiler with good standard conformance.

To effectively use the .Net framework, which Microsoft presents as the future of Windows, a language has to support a Java-like set of facilities. This implies support for a large number of language features including a massive metadata mechanism and inheritance — complete with covariant arrays (§4.1.1). In particular, a language used to produce components for consumption by other languages must produce complete metadata and a language that wants to consume components produced by other languages must be able to accept the metadata they produce. The Microsoft C++ dialect that supports all that, ISO C++ plus "The CLI extensions to ISO C++", colloquially referred to as C++/CLI [41], will obviously play a major role in the future of the C++ world. Interestingly, C++ with the C++/CLI extensions is the only language that provides access to every feature of .Net. Basically, C++/CLI is a massive set of extensions to ISO C++ and provides a degree of integration with Windows that makes it unlikely that a program that relies on C++/CLI features in any significant way will be portable beyond the Microsoft platforms that provide the massive infrastructure on which C++/CLI depends. As ever, systems interfaces can be encapsulated, and must be encapsulated to preserve portability. In addition to ISO C++, C++/CLI provides its own loop construct, overloading mechanisms (indexers), "properties", event mechanism, garbage-collected heap, a different class object initialization semantics, a new form of references, a new form of pointers, generics, a new set of standard containers (the .Net ones), and more.

In the early years of .Net (around 2000 onwards), Microsoft provided a dialect called "managed C++" [24]. It was generally considered "pretty lame" (if essential to some Microsoft users) and appears to have been mostly a stopgap measure — without any clear indication what the future might bring for its users. It has now been superseded by the much more comprehensive and carefully designed C++/CLI.

One of my major goals in working in the standards committee was to prevent C++ from fracturing into dialects. Clearly, in the case of C++/CLI, I and the committee failed. C++/CLI has been standardized by ECMA [41] as a binding to C++. However, Microsoft's main products, and those of their major customers, are and will remain in C++. This ensures good compiler and tool support for C++ — that is ISO C++ — under Windows for the foreseeable future. People who care about portability can program around the Microsoft extensions to ensure platform independence (§7.4). By default the Microsoft compiler warns about use of C++/CLI extensions.

The members of the C++ standards committee were happy to see C++/CLI as an ECMA standard. However, an attempt to promote that standard to an ISO standard caused a flood of dissent. Some national standards bodies — notably the UK's C++ Panel — publicly expressed serious concern [142]. This caused people at Microsoft to better document their design rationale [140] and to be more careful about not confusing ISO C++ and C++/CLI in Microsoft documentation.

7.9 Dialects

Obviously, not everyone who thought C++ to be basically a good idea wanted to go though the long and often frustrating ISO standards process to get their ideas into the mainstream. Similarly, some people consider compatibility overrated or even a very bad idea. In either case, people felt they could make faster progress and/or better by simply defining and implementing their own dialect. Some hoped that their dialect would eventually become part of the mainstream; others thought that life outside the mainstream was good enough for their purposes, and a few genuinely aimed at producing a short-lived language for experimental purposes.

There have been too many C++ dialects (many dozens) for me to mention more than a tiny fraction. This is not ill will — though I am no great fan of dialects because they fracture the user community [127, 133] — but a reflection that they have had very little impact outside limited communities. So far, no major language feature has been brought into the mainstream of C++ from a dialect. However, C++0x will get something that looks like C++/CLI's properly scoped `enums` [138], a C++/CLI-like `for`-statement [84], and the keyword `nullptr` [139] (which curiously enough was suggested by me for C++/CLI).

Concurrency seems to be an extremely popular area of language extension. Here are a few C++ dialects supporting some form of concurrency with language features:

- Concurrent C++[46]
- micro C++[18]
- ABC++ [83]
- Charm++ [75]
- POOMA [86]
- C++// [21]

Basically, every trend and fad in the computer science world spawned a couple of C++ dialects, such as Aspect C++ [100], R++ [82], Compositional C++ [25], Objective C++ [90], Open C++ [26], and dozens more. The main problem with these dialects is their number. Dialects tend to split up a sub-user community to the point where none reach a large enough user community to afford a sustainable infrastructure [133].

Another kind of dialect appears when a vendor adds some (typically minor) "neat features" to their compiler to help their users with some specific tasks (e.g. OS access and optimization). These become barriers to portability even if they are often beloved by some users. They are also appreciated by some managers and platform fanatics as a lock-in mechanism. Examples exist for essentially every implementation supplier; for example Borland (Delphi-style properties),GNU (variable and type attributes), and Microsoft (C++/CLI; §7.8). Such dialect features are nasty when they appear in header files, setting off a competitive scramble among vendors keen on being able to cope with their competitors' code. They also significantly complicate compiler, library, and tool building.

It is not obvious when a dialect becomes a completely separate language and many languages borrowed heavily from C++ (with or without acknowledgments) without aiming for any degree of compatibility. A recent example is the "D" language (the currently most recent language with that popular name): "D was conceived in December 1999 by Walter Bright as a reengineering of C and C++" [17].

Even Java started out (very briefly) as a C++ dialect [164], but its designers soon decided that maintaining compatibility with C++ would be too constraining for their needs. Since their aims included 100% portability of Java code and a restriction of code styles to a version of object-oriented programming, they were almost certainly correct in that. Achieving any degree of useful compatibility is very hard, as the C/C++ experience shows.

8. C++0x

From late 1997 until 2002, the standards committee deliberately avoided serious discussion of language extension. This allowed compiler, tool, and library implementers to catch up and users to absorb the programming techniques supported by Standard C++. I first used the term "C++0x" in 2000 and started a discussion of "directions for C++0x" through presentations in the committee and elsewhere from 2001 onwards. By 2003, the committee again considered language extensions. The "extensions working group" was reconstituted as the "evolution working group". The name change (suggested by Tom Plum) is meant to reflect a greater emphasis on the integration of language features and standard library facilities. As ever, I am the chairman of that working group, hoping to help ensure a continuity of vision for C++ and a coherence of the final result. Similarly, the committee membership shows continuous participation of a large number of people and organizations. Fortunately, there are also many new faces bringing new interests and new expertise to the committee.

Obviously, C++0x is still work in progress, but years of work are behind us and and many votes have been taken. These votes are important in that they represent the response of an experienced part of the C++ community to the problems with C++98 and the current challenges facing C++ programmers. The committee intended to be cautious and conservative about changes to the language itself, and strongly emphasize compatibility. The stated aim was to channel the major effort into an expansion of the standard library. In the standard library, the aim was to be aggressive and opportunistic. It is proving difficult to deliver on that aim, though. As usual, the committee just doesn't have sufficient resources. Also, people seem to get more excited over language extensions and are willing to spend more time lobbying for and against those.

The rate of progress is roughly proportional to the number of meetings. Since the completion of the 1998 standard, there has been two meetings a year, supplemented by a large amount of web traffic. As the deadlines for C++0x approach, these large meetings are being supplemented by several "extra" meetings focused on pressing topics, such as concepts (§8.3.3) and concurrency (§5.5).

8.1 Technical Challenges

What technical challenges faced the C++ community at the time when C++0x was beginning to be conceived? At a high level an answer could be:

- GUI-based application building
- Distributed computing (especially the web)
- Security

The big question is how to translate that into language features, libraries (ISO standard or not), programming environment, and tools. In 2000-2002, I tried unsuccessfully to get the standards committee's attention on the topic of distributed computing and Microsoft representatives regularly try to raise the topic of security. However, the committee simply doesn't think like that. To make progress, issues have to be expressed as concrete proposals for change, such as to add a (specific) language feature to support callbacks or to replace the notoriously unsafe C standard library function `gets()` with a (specific) alternative. Also, the C++ tradition is to approach challenges obliquely though improved abstraction mechanisms and libraries rather than providing a solution for a specific problem.

After I had given a couple of talks on principles and directions, the first concrete action in the committee was a brainstorming session at the 2001 meeting in Redmond, Washington. Here we made a "wish list" for features for C++0x. The first suggestion was for a portable way of expressing alignment constraints (by P. J. Plauger supported by several other members). Despite my strongly expressed urge to focus on libraries, about 90 out of about 100 suggestions were for language features. The saving grace was that many of those were for features that would ease the design, implementation, and use of more elegant, more portable, and efficient libraries. This brainstorming provided the seeds of maintained "wish lists" for C++0x language features and standard library facilities [136].

From many discussions and presentations, I have come with a brief summary of general aims and design rules for C++0x that appear to be widely accepted. Aims:

- Make C++ a better language for systems programming and library building — rather than providing specialized facilities for a particular sub-community (e.g. numeric computation or Windows-style application development)
- Make C++ easier to teach and learn — through increased uniformity, stronger guarantees, and facilities supportive of novices (there will always be more novices than experts)

Rules of thumb:

- Provide stability and compatibility
- Prefer libraries to language extensions
- Make only changes that change the way people think
- Prefer generality to specialization
- Support both experts and novices
- Increase type safety
- Improve performance and ability to work directly with hardware
- Fit into the real world

These lists have been useful as a framework for rationales for proposed extensions. Dozens of committee technical papers have used them. However, they provide only a weak set of directions and are only a weak counterweight to a widespread tendency to focus on details. The GUI and distributed computing challenges are not directly represented here, but feature prominently under “libraries” and “work directly with hardware” (§8.6).

8.2 Suggested Language Extensions

To give a flavor of the committee work around 2005, consider a (short!) incomplete list of proposed extensions:

1. `decltype` and `auto` — type deduction from expressions (§8.3.2)
2. Template aliases — a way of binding some but not all template parameters and naming the resulting partial template specialization
3. Extern templates — a way of suppressing implicit instantiation in a translation unit
4. Move semantics (rvalue references) — a general mechanism for eliminating redundant copying of values
5. Static assertions (`static_assert`)
6. `long long` and many other C99 features
7. >> (without a space) to terminate two template specializations
8. Unicode data types
9. Variadic templates (§8.3.1)
10. Concepts — a type system for C++ types and integer values (§8.3.3)
11. Generalized constant expressions [39]
12. Initializer lists as expressions (§8.3.1)
13. Scoped and strongly typed enumerations [138]
14. Control of alignment
15. `nullptr` — Null pointer constant [139]
16. A `for`-statement for ranges
17. Delegating constructors
18. Inherited constuctors
19. Atomic operations
20. Thread-local storage
21. Defaulting and inhibiting common operations
22. Lambda functions
23. Programmer-controlled garbage collection [12] (§5.4)
24. In-class member initializers
25. Allow local classes as template parameters
26. Modules
27. Dynamic library support
28. Integer sub-ranges
29. Multi-methods
30. Class namespaces
31. Continuations
32. Contract programming — direct support for preconditions, postconditions, and more
33. User-defined operator. (dot)
34. `switch` on `string`
35. Simple compile-time reflection
36. `#nomacro` — a kind of scope to protect code from unintended macro expansion
37. GUI support (e.g., slots and signals)
38. Reflection (i.e., data structures describing types for run-time use)
39. concurrency primitives in the language (not in a library)

As ever, there are far more proposals than the committee could handle or the language could absorb. As ever, even accepting all the good proposals is infeasible. As ever, there seems to be as many people claiming that the committee is spoiling the language by gratuitous complicated features as there are people who complain that the committee is killing the language by refusing to accept essential features. If you take away consistent overstatement of arguments, both sides have a fair degree of reason behind them. The balancing act facing the committee is distinctly nontrivial.

As of October 2008, Items 1-7 have been approved. Based on the state of proposals and preliminary working group votes, my guess is that items 10-21 will also be accepted. Beyond that, it’s hard to guess. Proosals 22-25 are being developed aiming for votes in July 2007 and proposal 26 (modules) has been postponed to a technical report.

Most of these are being worked upon under one or more of the “rules of thumb” listed above. That list is less than half of the suggestions that the committee has received in a form that compels it to (at least briefly) consider them. My collection of suggestions from emails and meetings with users is several times that size. At my urging, the committee at the spring 2005 meeting in Lillehammer, Norway decided (by vote) to stop accepting new proposals. In October of 2006, this vote was followed up by a vote to submit a draft standard in late 2007 so as to make C++0x into C++09. However, even with the stream of new proposals stemmed, it is obvious that making a coherent whole of a selection of features will be a practical challenge as well as a technical one.

To give an idea of the magnitude of the technical challenge, consider that a paper on part of the concepts problem was accepted for the premier academic conference in the field, POPL, in 2006 [38] and other papers analyzing problems related to concepts were presented at OOPSLA [44, 52] and PLDI [74]. I personally consider the technical problems related to the support of concurrency (including the memory model) harder still — and essential. The C++0x facilities for dealing with concurrency are briefly discussed in §8.6.

The practical challenge is to provide correct and consistent detailed specifications of all these features (and standard library facilities). This involves producing and checking hundreds of pages of highly technical standards text. More often than not, implementation and experimentation are part of the effort to ensure that a feature is properly specified and interacts well with other features in the language and the standard library. The standard specifies not just an implementation (like a vendor's documentation), but a whole set of possible implementations (different vendors will provide different implementations of a feature, each with the same semantics, but with different engineering tradeoffs). This implies an element of risk in accepting any new feature — even in accepting a feature that has already been implemented and used. Committee members differ in their perception of risk, in their views of what risks are worth taking, and in their views of how risk is best handled. Naturally, this is a source of many difficult discussions.

All proposals and all the meeting minutes of the committee are available on the committee's website [69]. That's more than 2000 documents — some long. A few of the pre-1995 papers are not yet (August 2006) available online because the committee relied on paper until about 1994.

8.3 Support for Generic Programming

For the language itself, we see an emphasis on features to support generic programming because generic programming is the area where use of C++ has progressed the furthest relative to the support offered by the language.

The overall aim of the language extensions supporting generic programming is to provide greater uniformity of facilities so as to make it possible to express a larger class of problems directly in a generic form. The potentially most significant extensions of this kind are:

- general initializer lists (§8.3.1)
- `auto` (§8.3.2)
- concepts (§8.3.3)

Of these proposals, only `auto` has been formally voted in. The others are mature proposals and initial "straw votes" have been taken in their support.

8.3.1 General initializer lists

"Provide as good support for user-defined types as for built-in types" is prominent among the language-technical design principles of C++ (§2). C++98 left a major violation of that principle untouched: C++98 provides notational support for initializing a (built-in) array with a list of values, but there is no such support for a (user-defined) `vector`. For example, we can easily define an array initialized to the three `int`s `1`, `2`, and `3`:

```
int a[] = { 1,2,3 };
```

Defining an equivalent `vector` is awkward and may require the introduction of an array:

```
// one way:
     vector v1;
     v1.push_back(1);
     v1.push_back(2);
     v1.push_back(3);

// another way
     int a[] = { 1,2,3 };
     vector v2(a,a+sizeof(a)/sizeof(int));
```

In C++0x, this problem will be remedied by allowing a user to define a "sequence constructor" for a type to define what initialization with an initializer list means. By adding a sequence constructor to `vector`, we would allow:

```
vector<int> v = { 1,2,3 };
```

Because the semantics of initialization defines the semantics of argument passing, this will also allow:

```
int f(const vector<int>&);
// ...
int x = f({ 1,2,3 });
int y = f({ 3,4,5,6,7,8, 9, 10 });
```

That is, C++0x gets a type-safe mechanism for variable-length homogeneous argument lists [135].

In addition, C++0x will support type-safe variadic template arguments [54] [55]. For example:

```
template<class ... T> void print(const T& ...);
// ...
string name = "World"'
print("Hello, ",name,'!');
int x = 7;
print("x = ",x);
```

At the cost of generating a unique instantiation (from a single template function) for each call with a given set of argument types, variadic templates allow arbitrary non-homogenous argument lists to be handled. This is especially useful for tuple types and similar abstractions that inherently deal with heterogeneous lists.

I was the main proponent of the homogenous initializer list mechanism. Doug Gregor and Jaakko Järvi and Doug Gregor designed the variadic template mechanism, which conceptually has its roots in Jakko Järvi and Gary Powell's lambda library [72] and the tuple library [71].

8.3.2 Auto

C++0x will support the notion of a variable being given a type deduced from the type of its initializer [73]. For example, we could write the verbose example from §4.1.3 like this:

```
auto q = find(vi.begin(),vi.end(),7); // ok
```

Here, we deduce the type of `q` to be the return type of the value returned by `find`, which in turn is the type of `vi.begin()`; that is, `vector<int>::iterator`. I first implemented that use of `auto` in 1982, but was forced to back it out of "C with Classes" because of a C compatibility problem. In K&R C [76] (and later in C89 and ARM C++), we can omit the type in a declaration. For example:

```
static x;    // means static int x
auto y;      // means stack allocated int y
```

After a proposal by Dag Bruc̆k, both C99 and C++98 banned this "implicit `int`". Since there is now no legal code for "`auto q`" to be incompatible with, we allowed it. That incompatibility was always more theoretical than real (reviews of huge amounts of code confirm that), but using the (unused) keyword `auto` saved us from introducting a new keyword. The obvious choice (from many languages, including BCPL) is `let`, but every short meaningful word has already been used in many programs and long and cryptic keywords are widely disliked.

If the committee — as planned — accepts overloading based on concepts (§8.3.3) and adjusts the standard library to take advantage, we can even write:

```
auto q = find(vi,7); // ok
```

in addition to the more general, but wordier:

```
auto q = find(vi.begin(),vi.end(),7); // ok
```

Unsurprisingly, given its exceptionally long history, I was the main designer of the `auto` mechanism. As with every C++0x proposal, many people contributed, notably Gabriel Dos Reis, Jaakko Järvi, and Walter Brown.

8.3.3 Concepts

The D&E [121] discussion of templates contains three whole pages (§15.4) on constraints on template arguments. Clearly, I felt the need for a better solution — and so did many others. The error messages that come from slight errors in the use of a template, such as a standard library algorithm, can be spectacularly long and unhelpful. The problem is that the template code's expectations of its template arguments are implicit. Consider again `find_if`:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
     while (first!=last && !pred(*first))
        ++first;
     return first;
}
```

Here, we make a lot of assumptions about the `In` and `Pred` types. From the code, we see that `In` must somehow support `!=`, `*`, and `++` with suitable semantics and that we must be able to copy `In` objects as arguments and return values. Similarly, we see that we can call a `Pred` with an argument of whichever type `*` returns from an `In` and apply `!` to the result to get something that can be treated as a `bool`. However, that's all implicit in the code. Note that a lot of the flexibility of this style of generic programming comes from implicit conversions used to make template argument types meet those requirements. The standard library carefully documents these requirements for input iterators (our `In`) and predicates (our `Pred`), but compilers don't read manuals. Try this error and see what your compiler says:

```
find_if(1,5,3.14);        // errors
```

On 2000-vintage C++ compilers, the resulting error messages were uniformly spectacularly bad.

Constraints classes A partial, but often quite effective, solution based on my old idea of letting a constructor check assumptions about template arguments (D&E §15.4.2) is now finding widespread use. For example:

```
template<class T> struct Forward_iterator {
     static void constraints(T a) {
        ++a; a++;        // can increment
        T b = a; b = a;  // can copy
        *b = *a;         // can dereference
                       // and copy result
   }
     Forward_iterator()
      { void (*p)(T) = constraints; }
};
```

This defines a class that will compile if and only if `T` is a forward iterator [128]. However, a `Forward_iterator` object doesn't really do anything, so that a compiler can (and all do) trivially optimize away such objects. We can use `Forward_iterator` in a definition like this:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
     Forward_iterator<In>();  // check
     while (first!=last && !pred(*first))
        ++first;
     return first;
}
```

Alex Stepanov and Jeremy Siek did a lot to develop and popularize such techniques. One place where they are used prominently is in the Boost library [16]. The difference in the quality of error messages can be spectacular. However, it is not easy to write constraints classes that consistently give good error messages on all compilers.

Concepts as a language feature Constraints classes are at best a partial solution. In particular, the type checking is done in the template definition. For proper separation of concerns, checking should rely only on information presented in a declaration. That way, we would obey the usual rules for interfaces and could start considering the possibility of genuine separate compilation of templates.

So, let's tell the compiler what we expect from a template argument:

```
template<ForwardIterator In, Predicate Pred>
In find_if(In first, In last, Pred pred);
```

Assuming that we can express what a ForwardIterator and a Predicate are, the compiler can now check a call of find_if in isolation from its definition. What we are doing here is to build a type system for template arguments. In the context of modern C++, such "types of types" are called *concepts* (see §4.1.8). There are various ways of specifying such concepts; for now, think of them as a kind of constraints classes with direct language support and a nicer syntax. A concept says what facilities a type must provide, but nothing about how it provides those facilities. The use of a concept as the type of a template argument (e.g. <ForwardIterator In>) is very close to a mathematical specification "for all types In such that an In can be incremented, dereferenced, and copied", just as the original <class T> is the mathematical "for all types T".

Given only that declaration (and not the definition) of find_if, we can write

```
int x = find_if(1,2,Less_than<int>(7));
```

This call will fail because int doesn't support unary * (dereference). In other words, the call will fail to compile because an int isn't a ForwardIterator. Importantly, that makes it easy for a compiler to report the error in the language of the user and at the point in the compilation where the call is first seen.

Unfortunately, knowing that the iterator arguments are ForwardIterators and that the predicate argument is a Predicate isn't enough to guarantee successful compilation of a call of find_if. The two argument types interact. In particular, the predicate takes an argument that is an iterator dereferenced by *: pred(*first). Our aim is complete checking of a template in isolation from the calls and complete checking of each call without looking at the template definition. So, "concepts" must be made sufficiently expressive to deal with such interactions among template arguments. One way is to parameterize the concepts in parallel to the way the templates themselves are parameterized. For example:

```
template<Value_type T,
   ForwardIterator<T> In,  // sequence of Ts
   Predicate<bool,T> Pred> // takes a T;
                           // returns a bool
   In find_if(In first, In last, Pred pred);
```

Here, we require that the ForwardIterator must point to elements of a type T, which is the same type as the Predicate's argument type. However, that leads to overly rigid interactions among template argument types and very complex patterns and redundant of parameterization [74]. The current concept proposal [129, 130, 132, 97, 53] focuses on expressing relationships among arguments directly:

```
template<ForwardIterator In,  // a sequence
         Predicate Pred>      // returns a bool
   requires Callable<Pred,In::value_type>
In find_if(In first, In last, Pred pred);
```

Here, we require that the In must be a ForwardIterator with a value_type that is acceptable as an argument to Pred which must be a Predicate.

A *concept* is a compile-time predicate on a set of types and integer values. The concepts of a single type argument provide a type system for C++ types (both built-in and user-defined types) [132].

Specifying template's requirements on its argument using concepts also allows the compiler to catch errors in the template definition itself. Consider this plausible pre-concept definition:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
     while(first!=last && !pred(*first))
        first = first+1;
     return first;
}
```

Test this with a vector or an array, and it will work. However, we specified that find_if should work for a ForwardIterator. That is, find_if should be usable for the kind of iterator that we can supply for a list or an input stream. Such iterators cannot simply move N elements forward (by saying p=p+N) — not even for N==1. We should have said ++first, which is not just simpler, but correct. Experience shows that this kind of error is very hard to catch, but concepts make it trivial for the compiler to detect it:

```
template<ForwardIterator In, Predicate Pred>
      requires Callable<Pred,In::value_type>
In find_if(In first, In last, Pred pred)
{
     while(first!=last && !pred(*first))
             first = first+1;
     return first;
}
```

The + operator simply isn't among the operators specified for a ForwardIterator. The compiler has no problems detecting that and reporting it succinctly.

One important effect of giving the compiler information about template arguments is that overloading based on the properties of argument types becomes easy. Consider again find_if. Programmers often complain about having to specify the beginning and the end of a sequence when all they want to do is to find something in a container. On the other hand, for generality, we need to be able to express algorithms in terms of sequences delimited by iterators. The obvious solution is to provide both versions:

```
template<ForwardIterator In, Predicate Pred>
   requires Callable<Pred,In::value_type>
In find_if(In first, In last, Pred pred);

template<Container C, Predicate Pred>
   requires Callable<Pred,C::value_type>
In find_if(C& c, Pred pred);
```

Given that, we can handle examples like the one in §8.3.2 as well as examples that rely on more subtle differences in the argument types.

This is not the place to present the details of the concept design. However, as presented above, the design appears to have a fatal rigidity: The expression of required properties of a type is often in terms of required member types. However, built-in types do not have members. For example, how can an `int*` be a `ForwardIterator` when a `ForwardIterator` as used above is supposed to have a member `value_type`? In general, how can we write algorithms with precise and detailed requirements on argument types and expect that authors of types will define their types with the required properties? We can't. Nor can we expect programmers to alter old types whenever they find a new use for them. Real-world types are often defined in ignorance of their full range of uses. Therefore, they often fail to meet the detailed requirements even when they possess all the fundamental properties needed by a user. In particular, `int*` was defined 30 years ago without any thought of C++ or the STL notion of an iterator. The solution to such problems is to (non-intrusively) map the properties of a type into the requirements of a concept. In particular, we need to say "when we use a pointer as a forward iterator we should consider the type of the object pointed to its `value_type`". In the C++0x syntax, that is expressed like this:

```
template<class T>
concept_map ForwardIterator<T*> {
     typedef T value_type;
}
```

A `concept_map` provides a map from a type (or a set of types; here, `T*`) to a `concept` (here, `ForwardIterator`) so that users of the concept for an argument will see not the actual type, but the mapped type. Now, if we use `int*` as a `ForwardIterator` that `ForwardIterator`'s `value_type` will be `int`. In addition to providing the appearance of members, a `concept_map` can be used to provide new names for functions, and even to provide new operations on objects of a type.

Alex Stepanov was probably the first to call for "concepts" as a C++ language feature [104] (in 2002) — I don't count the numerous vague and nonspecific wishes for "better type checking of templates". Initially, I was not keen on the language approach because I feared it would lead in the direction of rigid interfaces (inhibiting composition of separately developed code and seriously hurting performance), as in earlier ideas for language-supported "constrained genericity". In the summer of 2003, Gabriel Dos Reis and I analyzed the problem, outlined the design ideals, and documented the basic approaches to a solution [129] [130]. So, the current design avoids those ill effects (e.g., see [38]) and there are now many people involved in the design of concepts for C++0x, notably Doug Gregor, Jaakko Järvi, Gabriel Dos Reis, Jeremy Siek, and me. An experimental implementation of concepts has been written by Doug Gregor, together with a version of the STL using concepts [52].

I expect concepts to become central to all generic programming in C++. They are already central to much design using templates. However, existing code — not using concepts — will of course continue to work.

8.4 Embarrassments

My other priority (together with better support for generic programming) is better support for beginners. There is a remarkable tendency for proposals to favor the expert users who propose and evaluate them. Something simple that helps only novices for a few months until they become experts is often ignored. I think that's a potentially fatal design bias. Unless novices are sufficiently supported, only few will become experts. Bootstrapping is most important! Further, many — quite reasonably — don't want to become experts; they are and want to remain "occasional C++ users". For example, a physicist using C++ for physics calculations or the control of experimental equipment is usually quite happy being a physicist and has only limited time for learning programming techniques. As computer scientists we might wish for people to spend more time on programming techniques, but rather than just hoping, we should work on removing unnecessary barriers to adoption of good techniques. Naturally, most of the changes needed to remove "embarrassments" are trivial. However, their solution is typically constrained by compatibility concerns and concerns for the uniformity of language rules.

A very simple example is

```
vector<vector<double>> v;
```

In C++98, this is a syntax error because >> is a single lexical token, rather than two >s each closing a template argument list. A correct declaration of `v` would be:

```
vector< vector<double> > v;
```

I consider that an embarrassment. There are perfectly good language-technical reasons for the current rule and the extensions working group twice rejected my suggestions that this was a problem worth solving. However, such reasons are language technical and of no interest to novices (of all backgrounds — including experts in other languages). Not accepting the first and most obvious declaration of `v` wastes time for users and teachers. A simple solution of the ">> problem" was proposed by David Vandevoorde [145] and voted in at the 2005 Lillehammer meeting, and I expect many small "embarrassments" to be absent from C++0x. However, my attempt together with Francis Glassborow and others, to try to systematically eliminate the most frequently occurring such "embarrassments" seems to be going nowhere.

Another example of an "embarrassment" is that it is legal to copy an object of a class with a user-defined destructor using a default copy operation (constructor or assignment).

Requiring user-defined copy operations in that case would eliminate a lot of nasty errors related to resource management. For example, consider an oversimplified string class:

```
class String {
public:
     String(char* pp);           // constructor
     ~String() { delete[] pp; } // destructor
     char& operator[](int i);
private:
     int sz;
     char* p;
};

void f(char* x)
{
   String s1(x);
   String s2 = s1;
}
```

After the construction of `s2`, `s1.p` and `s2.p` point to the same memory. This memory (allocated by the constructor) will be deleted twice by the destructor, probably with disastrous results. This problem is obvious to the experienced C++ programmer, who will provide proper copy operations or prohibit copying. The problem has also been well documented from the earliest days of C++: The two obvious solutions can be found in TC++PL1 and D&E. However, the problem can seriously baffle a novice and undermine trust in the language. Language solutions have been proposed by Lois Goldtwaith, Francis Glassborow, Lawrence Crowl, and I [30]; some version may make it into C++0x.

I chose this example to illustrate the constraints imposed by compatibility. The problem could be eliminated by not providing default copy of objects of a class with pointer members; if people wanted to copy, they could supply a copy operator. However, that would break an unbelievable amount of code. In general, remedying long-standing problems is harder than it looks, especially if C compatibility enters into the picture. However, in this case, the existence of a destructor is a strong indicator that default copying would be wrong, so examples such as `String` could be reliably caught by the compiler.

8.5 Standard libraries

For the C++0x standard library, the stated aim was to make a much broader platform for systems programming. The June 2006 version "Standard Libraries Wish List" maintained by Matt Austern lists 68 suggested libraries including

- Container-based algorithms
- Random access to files
- Safe STL (completely range checked)
- File system access
- Linear algebra (vectors, matrices, etc.)
- Date and time
- Graphics library
- Data compression
- Unicode file names
- Infinite-precision integer arithmetic
- Uniform use of std::string in the library
- Threads
- Sockets
- Comprehensive support for unicode
- XML parser and generator library
- Graphical user interface
- Graph algorithms
- Web services (SOAP and XML bindings)
- Database support
- Lexical analysis and parsing

There has been work on many of these libraries, but most are postponed to post-C++0x library TRs. Sadly, this leaves many widely desired and widely used library components unstandardized. In addition, from observing people struggling with C++98, I also had high hopes that the committee would take pity on the many new C++ programmers and provide library facilities to support novices from a variety of backgrounds (not just beginning programmers and refugees from C). I have low expectations for the most frequently requested addition to the standard library: a standard GUI (Graphical User Interface; see §1 and §5.5).

The first new components of the C++0x standard library are those from the library TR (§6.2). All but one of the components were voted in at the spring 2006 meeting in Berlin. The special mathematical functions were considered to specialized for the vast majority of C++ programmers and were spun off to become a separate ISO standard.

In addition to the more visible work on adding new library components, much work in the library working group focuses on minor improvements to existing components, based on experience, and on improved specification of existing components. The accumulative effect of these minor improvements is significant.

The plan for 2007 includes going over the standard library to take advantage of new C++0x features. The first and most obvious example is to add rvalue initializers [57] (primarily the work of Howard Hinnant, Peter Dimov, and Dave Abrahams) to significantly improve the performance of heavily used components, such as `vector`, that occasionally have to move objects around. Assuming that concepts (§8.3.3) make it into C++0x, their use will revolutionize the specification of the STL part of the library (and other templated components). Similarly, the standard containers, such as `vector` should be augmented with sequence constructors to allow then to accept initializer lists (§8.3.1). Generalized constant expressions (`constexpr`, primarily the work of Gabriel Dos Reis and I [39]) will allow us to define simple functions, such as the ones defining properties of types in the `numeric_limits` and the `bitset` operators, so that they are usable at compile time. The variadic template mechanism (§8.3.1) dramatically simplifies interfaces to standard compunents, such as `tuple`, that can take a wide variety of template arguments. This has significant implications on the usability of these standard library components in performance-critical areas.

Beyond that, only a threads library seems to have gathered enough support to become part of C++0x. Other components that are likely to become part of another library TR (TR2) are:

- File system library — platform-independent file system manipulation [32]
- Date and time library [45]
- Networking — sockets, TCP, UDP, multicast, iostreams over TCP, and more [80]
- `numeric_cast` — checked conversions [22]

The initial work leading up to the proposals and likely votes mentioned here has been the main effort of the library working group 2003-06. The networking library has been used commercially for applications on multiple continents for some time. At the outset, Matt Austern (initially AT&T, later Apple) was the chair; now Howard Hinnant (initially Metrowerks, later Apple) has that difficult job.

In addition, the C committee is adopting a steady stream of technical reports, which must be considered and (despite the official ISO policy that C and C++ are distinct languages) will probably have to be adopted into the C++ library even though they — being C-style — haven't benefited from support from C++ language features (such as user-defined types, overloading, and templates). Examples are decimal floating-point and unicode built-in types with associated operations and functions.

All in all, this is a massive amount of work for the couple of dozen volunteers in the library working group, but it is not quite the "ambitious and opportunistic" policy that I had hoped for in 2001 (§8). However, people who scream for more (such as me) should note that even what's listed above will roughly double the size of the standard library. The process of library TRs is a hope for the future.

8.6 Concurrency

Concurrency cannot be perfectly supported by a library alone. On the other hand, the committee still considers language-based approaches to concurrency, such as is found in Ada and Java, insufficiently flexible (§5.5). In particular, C++0x must support current operating-system thread library approaches, such as POSIX threads and Windows threads.

Consequently, the work on concurrency is done by an ad hoc working group stradding the library-language divide. The approach taken is to carefully specify a machine model for C++ that takes into account modern hardware architectures [14] and to provide minimal language primitives:

- thread local storage [29]
- atomic types and operations [13]

The rest is left to a threads library. The current threads library draft is written by Howard Hinnant [58].

This "concurrency effort" is led by Hans Boehm (Hewlett-Packard) and Lawrence Crowl (formerly Sun, currently Google). Naturally, compiler writers and hardware vendors are most interested and supportive in this. For example, Clark Nelson from Intel redesigned the notion of sequencing that C++ inherited from C to better fit C++ on modern hardware [89]. The threads library will follow the traditions of Posix threads [19], Windows threads [163], and libraries based on those, such as boost::thread [16]. In particular, the C++0x threads library will not try to settle every issue with threading; instead it will provide a portable set of core facilities but leave some tricky issues system dependent. This is necessary to maintain compatibility both with earlier libraries and with the underlying operating systems. In addition to the typical thread library facilities (such as lock and mutex), the library will provide thread pools and a version of "futures" [56] as a higher-level and easier-to-use communications facility: A future can be described as a placeholder for a value to be computed by another thread; synchronization potentially happens when the future is read. Prototype implementations exist and are being evaluated [58].

Unfortunately, I had to accept that my hopes of direct support for distributed programming are beyond what we can do for C++0x.

9. Retrospective

This retrospective looks back with the hope of extracting lessons that might be useful in the future:

- Why did C++ succeed?
- How did the standards process serve the C++ community?
- What were the influences on C++ and what has been its impact?
- Is there a future for the ideals that underlie C++?

9.1 Why Did C++ Succeed?

That's a question thoughtful people ask me a few times every year. Less thoughtful people volunteer imaginative answers to it on the web and elsewhere most weeks. Thus, it is worth while to both briefly answer the question and to contradict the more popular myths. Longer versions of the answer can be found in D&E [121], in my first HOPL paper [120], and in sections of this paper (e.g., §1, §7, and §9.4). This section is basically a summary.

C++ succeeded because

- *Low-level access plus abstraction*: The original conception of C++, "C-like for systems programming plus Simula-like for abstraction," is a most powerful and useful idea (§7.1, §9.4).
- *A useful tool*: To be useful a language has to be complete and fit into the work environments of its users. It is neither necessary nor sufficient to be the best in the world at one or two things (§7.1).
- *Timing*: C++ was not the first language to support object-oriented programming, but from day one it was available as a useful tool for people to try for real-world problems.
- *Non-proprietary*: AT&T did not try to monopolize C++. Early on, implementations were relatively cheap (e.g., $750 for educational institutions) and in 1989 all rights to the language were transferred to the standard bodies (ANSI and later ISO). I and others from Bell Labs actively helped non-AT&T C++ implementers to get started.
- *Stable*: A high degree (but not 100%) of C compatibility was considered essential from day one. A higher degree of compatibility (but still not 100%) with earlier implementations and definitions was maintained throughout. The standards process was important here (§3.1, §6).
- *Evolving*: New ideas have been absorbed into C++ throughout (e.g., exceptions, templates, the STL). The world evolves, and so must a living language. C++ didn't just follow fashion; occationally, it led (e.g., genric programming and the STL). The standards process was important here (§4.1, §5, §8).

Personally, I particularly like that the C++ design never aimed as solving a clearly enumerated set of problems with a clearly enumerated set of specific solutions: I'd never design a tool that could only do what I wanted [116]. Developers tend not to appreciate this open-endedness and emphasis on abstraction until their needs change.

Despite frequent claims, the reason for C++'s success was *not*:

- *Just luck*: I am amazed at how often I hear that claimed. I fail to understand how people can imagine that I and others involved with C++ could — by pure luck — repeatedly have provided services preferred by millions of systems builders. Obviously an element of luck is needed in any endeavor, but to believe that mere bumbling luck can account for the use of C++ 1.0, C++ 2.0, ARM C++, and C++98 is stretching probabilities quite a bit. Staying lucky for 25 years can't be easy. No, I didn't imagine every use of templates or of the STL, but I aimed for generality (§4.1.2).
- *AT&T's marketing might*: That's funny! All other languages that have ever competed commercially with C++ were better financed. AT&T's C++ marketing budget for 1985-1988 (inclusive) was $5,000 (out of which we only managed to spend $3,000; see D&E). At the first OOPSLA, we couldn't afford to bring a computer, we had no flyers, and not a single marketer (we used a chalkboard, a signup sheet for research papers, and researchers). In later years, things got worse.
- *It was first*: Well, Ada, Eiffel, Objective C, Smalltalk and various Lisp dialects were all available commercially before C++. For many programmers, C, Pascal, and Modula-2 were serious contenders.
- *Just C compatibility*: C compatibility helped — as it should because it was a design aim. However, C++ was never just C and it earned the hostility from many C devotees early on for that. To meet C++'s design aims, some incompatibilities were introduced. C later adopted several C++ features, but the time lag ensured that C++ got more criticism than praise for that (§7.6). C++ was never an "anointed successor to C".
- *It was cheap*: In the early years, a C++ implementation was relatively cheap for academic institutions, but about as expensive as competitive languages for commercial use (e.g., about $40,000 for a commercial source license). Later, vendors (notably Sun and Microsoft) tended to charge more for C++ than for their own proprietary languages and systems.

Obviously, the reasons for success are as complex and varied as the individuals and organizations that adopted C++.

9.2 Effects of the Standards Process

Looking back on the '90s and the first years of the 21st century, what questions can we ask about C++ and the standards committee's stewardship of the language?

- Is C++98 better than ARM C++?
- What was the biggest mistake?
- What did C++ get right?
- Can C++ be improved? and if so, how?
- Can the ISO standards process be improved? and if so, how?

Yes, C++98 is a better language (with a much better library) than ARM C++ for the kinds of problems that C++ was designed to address. Thus, the efforts of the committee must be deemed successful. In addition, the population of C++ programmers has grown by more than an order of magnitude (from about 150,000 in 1990 to over 3 million in 2004; §1) during the stewardship of the committee. Could the committee have done better? Undoubtedly, but exactly how is hard to say. The exact conditions at the time where decisions are made are hard to recall (to refresh your memory, see D&E [121]).

So what was the biggest mistake? For the early years of C++, the answer was easy ("not shipping a larger/better foundation library with the early AT&T releases" [120]).

However, for the 1991-1998 period, nothing really stands out — at least if we have to be realistic. If we don't have to be realistic, we can dream of great support for distributed systems programming, a major library including great GUI support (§5.5), standard platform ABIs, elimination of overelaborate design warts that exist only for backward compatibility, etc. To get a serious candidate for regret, I think we have to leave the technical domain of language features and library facilities and consider social factors. The C++ community has no real center. Despite the major and sustained efforts of its members, the ISO C++ committee is unknown or incredibly remote to huge numbers of C++ users. Other programming language communities have been better at maintaining conferences, websites, collaborative development forums, FAQs, distribution sites, journals, industry consortiums, academic venues, teaching establishments, accreditation bodies, etc. I have been involved in several attempts to foster a more effective C++ community, but I must conclude that neither I nor anyone else have been sufficiently successful in this. The centrifugal forces in the C++ world have been too strong: Each implementation, library, and tool supplier has their own effort that usually does a good job for their users, but also isolates those users. Conferences and journals have fallen victim to economic problems and to a curious (but pleasant) lack of dogmatism among the leading people in the C++ community.

What did C++ get right? First of all, the ISO standard was completed in reasonable time and was the result of a genuine consensus. That's the base of all current C++ use. The standard strengthened C++ as a multi-paradigm language with uncompromising support for systems programming, including embedded systems programming.

Secondly, generic programming was developed and effective ways of using the language facilities for generic programming emerged. This not only involved work on templates (e.g., concepts) but also required a better understanding of resource management (§5.3) and of generic programming techniques. Bringing generic programming to the mainstream will probably be the main long-term technical contribution of C++ in this time period.

Can C++ be improved? and if so, how? Obviously, C++ can be improved technically, and I also think that it can be improved in the context of its real-world use. The latter is much harder, of course, because that imposes Draconian compatibility constraints and cultural requirements. The C++0x effort is the major attempt in that direction (§8). In the longer term (C++1x?), I hope for perfect type safety and a general high-level model for concurrency.

The question about improving the standards process is hard to answer. On the one hand, the ISO process is slow, bureaucratic, democratic, consensus-driven, subject to objections from very small communities (sometimes a single person), lacking of focus ("vision"), and cannot respond rapidly to changes in the industry or academic fashions. On the other hand, that process has been successful. I have sometimes summed up my position by paraphrasing Churchill: "the ISO standards process is the worst, except for all the rest". What the standards committee seems to lack is a "secretariat" of a handful of technical people who could spend full time examining needs, do comparative studies of solutions, experiment with language and library features, and work out detailed formulation of proposals. Unfortunately, long-time service in such a "secretariat" could easily be career death for a first-rate academic or industrial researcher. On the other hand, such a secretariat staffed by second-raters would lead to massive disaster. Maybe a secretariat of technical people serving for a couple of years supporting a fundamentally democratic body such as the ISO C++ committee would be a solution. That would require stable funding, though, and standards organizations do not have spare cash.

Commercial vendors have accustomed users to massive "standard libraries". The ISO standards process — at least as practiced by the C and C++ committees — has no way of meeting user expectations of a many hundred thousands of lines of free standard-library code. The Boost effort (§4.2) is an attempt to address this. However, standard libraries have to present a reasonably coherent view of computation and storage. Developers of such libraries also have to spend a huge amount of effort on important utility libraries of no significant intellectual interest: just providing "what everybody expects". There are also a host of mundane specification issues that even the commercial suppliers usually skimp on — but a standard must address because it aims to support multiple implementations. A loosely connected group of volunteers (working nights) is ill equipped to deal with that. A multi-year mechanism for guidance is necessary for coherence (and for integrating novel ideas) as well as ways of getting "ordinary development work" done.

9.3 Influences and impact

We can look at C++ in many ways. One is to consider influences:

1. What were the major influences on C++?
2. What languages, systems, and techniques were influenced by C++?

Recognizing the influences on the C++ language features is relatively easy. Recognizing the influences on C++ programming techniques, libraries, and tools is far harder because there is so much more going on in those areas and the decisions are not channeled through the single point of the standards committee. Recognizing C++'s influences on other languages, libraries, tools, and techniques is almost impossible. This task is made harder by a tendency of intellectual and commercial rivalries to lead to a lack of emphasis on documenting sources and influences. The competition for market share and mind share does not follow the ideal rules of academic publishing. In particular, most major ideas have multiple variants and sources, and people tend to refer to sources that are not seen as competitors — and for many new languages C++ is "the one to beat".

9.3.1 Influences on C++

The major influences on early C++ were the programming languages C and Simula. Along the way further influences came from Algol68, BCPL, Clu, Ada, ML, and Modula-2+ [121]. The language-technical borrowings tend to be accompanied by programming techniques related to those features. Many of the design ideas and programming techniques came from classical systems programming and from UNIX. Simula contributed not just language features, but ideas of programming technique relating to object-oriented programming and data abstraction. My emphasis on strong static type checking as a tool for design, early error detection, and run-time performance (in that order) reflected in the design of C++ came primarily from there. With generic programming and the STL, mainstream C++ programming received a solid dose of functional programming techniques and design ideas.

9.3.2 Impact of C++

C++'s main contribution was and is through the many systems built using it (§7 [137]). The primary purpose of a programming language is to help build good systems. C++ has become an essential ingredient in the infrastructure on which our civilization relies (e.g. telecommunications systems, personal computers, entertainment, medicine, and electronic commerce) and has been part of some of the most inspiring achievements of this time period (e.g., the Mars Rovers and the sequencing of the human genome).

C++ was preeminent in bringing object-oriented programming into the mainstream. To do so, the C++ community had to overcome two almost universal objections:

- Object-oriented programming is inherently inefficient
- Object-oriented programming is too complicated to be used by "ordinary programmers"

C++'s particular variant of OO was (deliberately) derived from Simula, rather than from Smalltalk or Lisp, and emphasized the role of static type checking and its associated design techniques. Less well recognized is that C++ also brought with it some non-OO data abstraction techniques and an emphasis on statically typed interfaces to non-OO facilities (e.g., proper type checking for plain old C functions; §7.6). When that was recognized, it was often criticized as "hybrid", "transitory", or "static". I consider it a valuable and often necessary complement to the view of object-oriented programming focused on class hierarchies and dynamic typing.

C++ brought generic programming into the mainstream. This was a natural evolution of the early C++ emphasis on statically typed interfaces and brought with it a number of functional programming techniques "translated" from lists and recursion to general sequences and iteration. Function templates plus function objects often take the role of higher-order functions. The STL was a crucial trendsetter. In addition, the use of templates and function objects led to an emphasis on static type safety in high-performance computing (§7.3) that hitherto had been missing in real-world applications.

C++'s influence on specific language features is most obvious in C, which has now "borrowed":

- function prototypes
- `const`
- `inline`
- `bool`
- `complex`
- declarations as statements
- declarations in for-statement initializers
- `//` comments

Sadly, with the exception of the `//` comments (which I in turn borrowed from BCPL), these features were introduced into C in incompatible forms. So was `void*`, which was a joint effort between Larry Rosler, Steve Johnson, and me at Bell Labs in mid-1982 (see D&E [121]).

Generic programming, the STL, and templates have also been very influential on the design of other languages. In the 1980s and 1990s templates and the programming techniques associated with them were frequently scorned as "not object-oriented", too complicated, and too expensive; workarounds were praised as "simple". The comments had an interesting similarity to the early community comments on object-oriented programming and classes. However, by 2005, both Java and C# have acquired "generics" and statically typed containers. These generics look a lot like templates, but Java's are primarily syntactic sugar for abstract classes and far more rigid than templates. Though still not as flexible or efficient as templates, the C# 2.0 generics are better integrated into the type system and even (as templates always did) offer a form of specialization.

C++'s success in real-world use also led to influences that are less beneficial. The frequent use of the ugly and irregular C/C++ style of syntax in modern languages is not something I'm proud of. It is, however, an excellent indicator of C++ influence — nobody would come up with such syntax from first principles.

There is obvious C++ influence in Java, C#, and various scripting languages. The influence on Ada95, COBOL, and Fortran is less obvious, but definite. For example, having an Ada version of the Booch components [15] that could compare in code size and performance with the C++ version was a commonly cited goal. C++ has even managed to influence the functional programming community (e.g. [59]). However, I consider those influences less significant.

One of the key areas of influence has been on systems for intercommunication and components, such as CORBA and COM, both of which have a fundamental model of interfaces derived from C++'s abstract classes.

9.4 Beyond C++

C++ will be the mainstay of much systems development for years to come. After C++0x, I expect to see C++1x as the language and its community adapts to new challenges. However, where does C++ fit philosophically among current languages; in other words, ignoring compatibility, what is the essence of C++? C++ is an approximation to ideals, but obviously not itself the ideal. What key properties, ideas, and ideals might become the seeds of new languages and programming techniques? C++ is

- a set of low-level language mechanisms (dealing directly with hardware)
- combined with powerful compositional abstraction mechanisms
- relying on a minimal run-time environment.

In this, it is close to unique and much of its strength comes from this combination of features. C is by far the most successful traditional systems programming language; the dominant survivor of a large class of popular and useful languages. C is close to the machine in exactly the way C++ is (§2, §6.1), but C doesn't provide significant abstraction mechanisms. To contrast, most "modern languages" (such as Java, C#, Python, Ruby, Haskell, and ML) provide abstraction mechanisms, but deliberately put barriers between themselves and the machine. The most popular rely on a virtual machine plus a huge run-time support system. This is a great advantage when you are building an application that requires the services offered, but a major limitation on the range of application areas (§7.1). It also ensures that every application is a part of a huge system, making it hard to understand completely and impossible to make correct and well performing in isolation.

In contrast, C++ and its standard library can be implemented in itself (the few minor exceptions to this are the results of compatibility requirements and minor specification mistakes). C++ is complete both as a mechanism for dealing with hardware and for efficient abstraction from the close-to-hardware levels of programming. Occasionally, optimal use of a hardware feature requires a compiler intrinsic or an inline assembler insert, but that does not break the fundamental model; in fact, it can be seen as an integral part of that model. C++'s model of computation is that of hardware, rather than a mathematical or ad hoc abstraction.

Can such a model be applied to future hardware, future systems requirements, and future application requirements? I consider it reasonably obvious that in the absence of compatibility requirements, a new language on this model can be much smaller, simpler, more expressive, and more amenable to tool use than C++, without loss of performance or restriction of application domains. How much smaller? Say 10% of the size of C++ in definition and similar in front-end compiler size. In the "Retrospective" chapter of D&E [121], I expressed that idea as "Inside C++, there is a much smaller and cleaner language struggling to get out". Most of the simplification would come from generalization — eliminating the mess of special cases that makes C++ so hard to handle — rather than restriction or moving work from compile time to run time. But could a machine-near plus zero-overhead abstraction language meet "modern requirements"? In particular, it must be

- completely type safe
- capable of effectively using concurrent hardware

This is not the place to give a technical argument, but I'm confident that it could be done. Such a language would require a realistic (relative to real hardware) machine model (including a memory model) and a simple object model. The object model would be similar to C++'s: direct mapping of basic types to machine objects and simple composition as the basis for abstraction. This implies a form of pointers and arrays to support a "sequence of objects" basic model (similar to what the STL offers). Getting that type safe — that is, eliminating the possibility of invalid pointer use, range errors, etc. — with a minimum of run-time checks is a challenge, but there is plenty of research in software, hardware, and static analysis to give cause for optimism. The model would include true local variables (for user-defined types as well as built-in ones), implying support for "resource acquisition is initialization"-style resource management. It would not be a "garbage collection for everything" model, even though there undoubtedly would be a place for garbage collection in most such languages.

What kind of programming would this sort of language support? What kind of programming would it support better than obvious and popular alternatives? This kind of language would be a systems programming language suitable for hard real-time applications, device drivers, embedded devices, etc. I think the ideal systems programming language belongs to this general model — you need machine-near features, predictable performance, (type) safety, and powerful abstraction features. Secondly, this kind of language would be ideal for all kinds of resource-constrained applications and applications with large parts that fitted these two criteria. This is a huge and growing class of applications. Obviously, this argument is a variant of the analysis of C++'s strengths from §7.1.

What would be different from current C++? Size, type safety, and integral support for concurrency would be the most obvious differences. More importantly, such a language would be more amenable to reasoning and to tool use than C++ and languages relying on lots of run-time support. Beyond that, there is ample scope for improvement over C++

in both design details and specification techniques. For example, the integration between core language and standard library features can be much smoother; the C++ ideal of identical support for built-in and user-defined types could be completely achieved so that a user couldn't (without examining the implementation) tell which was which. It is also obvious that essentially all of such a language could be defined relative to an abstract machine (thus eliminating all incidental "implementation defined" behavior) without compromising the close-to-machine ideal: machine architectures have huge similarities in operations and memory models that can be exploited.

The real-world challenge to be met is to specify and implement correct systems (often under resource constraints and often in the presence of the possibility of hardware failure). Our civilization critically depends on software and the amount and complexity of that software is steadily increasing. So far, we (the builders of large systems) have mostly addressed that challenge by patch upon patch and incredible numbers of run-time tests. I don't see that approach continuing to scale. For starters, apart from concurrent execution, our computers are not getting faster to compensate for software bloat as they did for the last decades. The way to deal that I'm most optimistic about is a more principled approach to software, relying on more mathematical reasoning, more declarative properties, and more static verification of program properties. The STL is an example of a move in that direction (constrained by the limitations of C++). Functional languages are examples of moves in that direction (constrained by a fundamental model of memory that differs from the hardware's and underutilization of static properties). A more significant move in that direction requires a language that supports specification of desired properties and reasoning about them (concepts are a step in that direction). For this kind of reasoning, run-time resolution, run-time tests, and multiple layers of run-time mapping from code to hardware are at best wasteful and at worst serious obstacles. The ultimate strength of a machine-near, type-safe, and compositional abstraction model is that it provides the fewest obstacles to reasoning.

10. Acknowledgments

Obviously, I owe the greatest debt to the members of the C++ standards committee who worked — and still work — on the language and library features described here. Similarly, I thank the compiler and tools builders who make C++ a viable tool for an incredible range of real-world problems.

Thanks to Matt Austern, Paul A. Bistow, Steve Clamage, Greg Colvin, Gabriel Dos Reis, S. G. Ganesh, Lois Goldthwaithe, Kevlin Henney, Howard Hinnant, Jaakko Järvi, Andy Koenig, Bronek Kozicki, Paul McJones, Scott Meyers, W. M. (Mike) Miller, Sean Parent, Tom Plum, Premanan M. Rao, Jonathan Schilling, Alexander Stepanov, Nicholas Stroustrup, James Widman, and J. C. van Winkel for making constructive comments on drafts of this paper.

Also thanks to the HOPL-III reviewers: Julia Lawall, Mike Mahoney, Barbara Ryder, Herb Sutter, and Ben Zorn who — among other things — caused a significant increase in the introductory material, thus hopefully making this paper more self-contained and more accessible to people who are not C++ experts.

I am very grateful to Brian Kernighan and Doug McIlroy for all the time they spent listening to and educating me on a wide variety of language-related topics during the 1990s. A special thanks to Al Aho of Columbia University for lending me an office in which to write the first draft of this paper.

References

[1] David Abrahams: *Exception-Safety in Generic Components*. M. Jazayeri, R. Loos, D. Musser (eds.): Generic Programming, Proc. of a Dagstuhl Seminar. Lecture Notes on Computer Science. Volume 1766. 2000. ISBN: 3-540-41090-2.

[2] David Abrahams and Aleksey Gurtovoy: *C++ Template Meta-programming* Addison-Wesley. 2005. ISBN 0-321-22725-5.

[3] Andrei Alexandrescu: *Modern C++ Design*. Addison-Wesley. 2002. ISBN 0-201-70431.

[4] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, Lawrence Rauchwerger: *STAPL: An Adaptive, Generic Parallel C++ Library*. In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), pp. 193-208, Cumberland Falls, Kentucky, Aug 2001.

[5] AT&T C++ translator release notes. *Tools and Reusable Components*. 1989.

[6] M. Austern: *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison Wesley. 1998. ISBN: 0-201-30956-4.

[7] John Backus: *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs*. Communications of the ACM 21, 8 (Aug. 1978).

[8] J. Barreiro, R. Fraley, and D. Musser: *Hash Tables for the Standard Template Library*. Rensselaer Polytechnic Institute and Hewlett Packard Laboratories. February, 1995. `ftp://ftp.cs.rpi.edu/pub/stl/hashdoc.ps.Z`

[9] Graham Birtwistle et al.: *SIMULA BEGIN*. Studentlitteratur. Lund. Sweden. 1979. ISBN 91-44-06212-5.

[10] Jasmin Blanchette and Mark Summerfield: *C++ GUI Programming with Qt3*. Prentice Hall. 2004. ISBN 0-13-124072-2.

[11] Hans-J. Boehm: *Space Efficient Conservative Garbage Collection*. Proc. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation. ACM SIGPLAN Notices. June 1993. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[12] Hans-J. Boehm and Michael Spertus: *Transparent Garbage Collection for C++*. ISO SC22 WG21 TR NN1943==06-

0013.

[13] Hans-J. Boehm: *An Atomic Operations Library for C++*. ISO SC22 WG21 TR N2047==06-0117.

[14] Hans-J. Boehm: *A Less Formal Explanation of the Proposed C++ Concurrency Memory Model*. ISO SC22 WG21 TR N2138==06-0208.

[15] Grady Booch: *Software Components with Ada*. Benjamin Cummings. 1988. ISBN 0-8053-0610-2.

[16] The Boost collection of libraries. `http://www.boost.org`.

[17] Walter Bright: *D Programming Language*. `http://www.digitalmars.com/d/`.

[18] Peter A. Buhr and Glen Ditchfield: *Adding Concurrency to a Programming Language*. Proc. USENIX C++ Conference. Portland, OR. August 1992.

[19] David Butenhof: *Programming With Posix Threads*. ISBN 0-201-63392-2. 1997.

[20] T. Cargill: *Exception handling: A False Sense of Security*. The C++ Report, Volume 6, Number 9, November-December 1994.

[21] D. Caromel et al.: *C++//*. In *Parallel programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.

[22] Fernando Cacciola: *A proposal to add a general purpose ranged-checked numeric_cast*<> (Revision 1). ISO SC22 WG21 TR N1879==05-0139.

[23] CGAL: Computational Geometry Algorithm Library. *http://www.cgal.org/*.

[24] Siva Challa and Artur Laksberg: *Essential Guide to Managed Extensions for C++*. Apress. 2002. ISBN: 1893115283.

[25] K. M. Chandy and C. Kesselman: *Compositional C++: Compositional Parallel Programming*. Technical Report. California Institute of Technology. [Caltech CSTR: 1992.cs-tr-92-13].

[26] Shigeru Chiba: *A Metaobject Protocol for C++*. Proc. OOPSLA'95. `http://www.csg.is.titech.ac.jp/~chiba/openc++.html`.

[27] Steve Clamage and David Vandevoorde. Personal communications. 2005.

[28] J. Coplien: *Multi-paradigm Design for C++*. Addison Wesley. 1998. ISBN 0-201-82467-1.

[29] Lawrence Crowl: *Thread-Local Storage*. ISO SC22 WG21 TR N1966==06-0036.

[30] Lawrence Crowl: *Defaulted and Deleted Functions*. ISO SC22 WG21 TR N2210==07-0070.

[31] Krzysztof Czarnecki and Ulrich W. Eisenecker: *Generative Programming — Methods, Tools, and Applications*. Addison-Wesley, June 2000. ISBN 0-201-30977-7.

[32] Beman Dawes: *Filesystem Library Proposal for TR2 (Revision 2)* . ISO SC22 WG21 TR N1934==06-0004.

[33] Jeffrey Dean and Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[34] D. Detlefs: *Garbage collection and run-time typing as a C++ library*. Proc. USENIX C++ conference. 1992.

[35] M. Ellis and B. Stroustrup: *The Annotated C++ Reference Manual* ("The ARM") Addison Wesley. 1989.

[36] Dinkumware: *Dinkum Abridged Library* `http://www.dinkumware.com/libdal.html`.

[37] Gabriel Dos Reis and Bjarne Stroustrup: *Formalizing C++*. TAMU CS TR. 2005.

[38] Gabriel Dos Reis and Bjarne Stroustrup: *Specifying C++ Concepts*. Proc. ACM POPL 2006.

[39] Gabriel Dos Reis and Bjarne Stroustrup: *Generalized Constant Expressions* (Revision 3). ISO SC22 WG21 TR N1980==06-0050.

[40] The Embedded C++ Technical Committee: *The Language Specification & Libraries Version*. WP-AM-003. Oct 1999 (`http://www.caravan.net/ec2plus/`).

[41] Ecma International: *ECMA-372 Standard: C++/CLI Language Specification*. `http://www.ecma-international.org/publications/standards/Ecma-372.htm`. December 2005.

[42] Boris Fomitch: *The STLport Story*. `http://www.stlport.org/doc/story.html`.

[43] Eric Gamma, et al.: *Design Patterns*. Addison-Wesley. 1994. ISBN 0-201-63361-2.

[44] R. Garcia, et al.: *A comparative study of language support for generic programming*. ACM OOPSLA 2003.

[45] Jeff Garland: *Proposal to Add Date-Time to the C++ Standard Library*. ISO SC22 WG21 TR N1900=05-0160.

[46] N.H. Gehani and W.D. Roome: *Concurrent C++: Concurrent Programming with Class(es)*. Software—Practice and Experience, 18(12):1157—1177, December 1988.

[47] Geodesic: Great circle. Now offered by Veritas.

[48] M. Gibbs and B. Stroustrup: *Fast dynamic casting*. Software—Practice&Experience. 2005.

[49] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir: *MPI: The Complete Reference — 2nd Edition: Volume 2 — The MPI-2 Extensions* . The MIT Press. 1998.

[50] Keith E. Gorlen: *An Object-Oriented Class Library for C++ Programs*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.

[51] J. Gosling and H. McGilton: *The Java(tm) Language Environment: A White Paper*. `http://java.sun.com/docs/white/langenv/`.

[52] Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Gabriel Dos Reis, Bjarne Stroustrup, and Andrew Lumsdaine: *Concepts: Linguistic Support for Generic Programming*. Proc. of ACM OOPSLA'06. October 2006.

[53] D. Gregor and B. Stroustrup: *Concepts*. ISO SC22 WG21 TR N2042==06-0012.

[54] D. Gregor, J. Järvi, G. Powell: *Variadic Templates* (Revision

3). ISO SC22 WG21 TR N2080==06-0150.

[55] Douglas Gregor and Jaakko Järvi: *Variadic Templates for C++*. Proc. 2007 ACM Symposium on Applied Computing. March 2007.

[56] R. H. Halstead: *MultiLisp: A Language for Concurrent Symbolic Computation*. TOPLAS. October 1985.

[57] H. Hinnant, D. Abrahams, and P. Dimov: *A Proposal to Add an Rvalue Reference to the C++ Language*. ISO SC22 WG21 TR N1690==04-0130.

[58] Howard E. Hinnant: *Multithreading API for C++0X - A Layered Approach*. ISO SC22 WG21 TR N2094==06-0164.

[59] J. Hughes and J. Sparud: *Haskell++: An object-oriented extension of Haskell*. Proc. Haskell Workshop, 1995.

[60] IA64 C++ ABI. `http://www.codesourcery.com/cxx-abi`.

[61] IDC report on programming language use. 2004. http://www.idc.com.

[62] *Standard for the C Programming Language*. ISO/IEC 9899. ("C89").

[63] *Standard for the C++ Programming Language*. ISO/IEC 14882. ("C++98").

[64] *Standard for the C Programming Language*. ISO/IEC 9899:1999. ("C99").

[65] International Organization for Standards: *The C Programming Language*. ISO/IEC 9899:2002. Wiley 2003. ISBN 0-470-84573-2.

[66] International Organization for Standards: *The C++ Programming Language* ISO/IEC 14882:2003. Wiley 2003. ISBN 0-470-84674-7.

[67] *Technical Report on C++ Performance*. ISO/IEC PDTR 18015.

[68] *Technical Report on C++ Standard Library Extensions*. ISO/IEC PDTR 19768.

[69] ISO SC22/WG21 website: `http://www.open-std.org/jtc1/sc22/wg21/`.

[70] Sorin Istrail and 35 others: *Whole-genome shotgun assembly and comparison of human genome assemblies*. Proc. National Academy of Sciences. February, 2004. `http://www.pantherdb.org/`.

[71] Jaakko Järvi: *Proposal for adding tuple type into the standard library*. ISO SC22 WG21 TR N1382==02-0040.

[72] Jaakko Järvi, Gary Powell, and Andrew Lumsdaine: *The Lambda Library: Unnamed Functions in C++*. Software-Practice and Experience, 33:259-291, 2003.

[73] J. Järvi, B. Stroustrup and G. Dos Reis: *Deducing the type of a variable from its initializer expression*. ISO SC22 WG21 TR N1894, Oct. 2005.

[74] Jaakko Järvi, Douglas Gregor, Jeremiah Willcock, Andrew Lumsdaine, and Jeremy Siek: *Algorithm specialization in generic programming: challenges of constrained generics in C++*. Proc. PLDI 2006.

[75] L. V. Kale and S. Krishnan: *CHARM++* in *Parallel programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.

[76] Brian Kernighan and Dennis Richie: *The C Programming Language* ("K&R"). Prentice Hall. 1978. ISBN 0-13-110163-3.

[77] Brian Kernighan and Dennis Richie: *The C Programming Language (2nd edition)* ("K&R2" or just "K&R"). Prentice Hall. 1988. ISBN 0-13-110362-8.

[78] Brian Kernighan: *Why Pascal isn't my favorite programming language*. AT&T Bell Labs technical report No. 100. July 1981.

[79] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++(revised)*. Proc. USENIX C++ Conference. San Francisco, CA. April 1990. Also, Journal of Object-Oriented Programming. July 1990.

[80] Christopher Kohlhoff: *Networking Library Proposal for TR2*. ISO SC22 WG21 TR N2054==06-0124.

[81] Mark A. Linton and Paul R. Calder: *The Design and Implementation of InterViews*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.

[82] A. Mishra et al.: R++: *Using Rules in Object-Oriented Designs*. Proc. OOPSLA-96. `http://www.research.att.com/sw/tools/r++/`.

[83] W. G. O'Farrell et al.: *ABC++* in *Parallel Programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.

[84] Thorsten Ottosen: *Proposal for new for-loop*. ISO SC22 WG21 TR N1796==05-0056.

[85] Sean Parent: personal communications. 2006.

[86] J. V. W. Reynolds et al.: *POOMA* in *Parallel Programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.

[87] Mike Mintz and Robert Ekendahl: *Hardware Verification with C++ — A Practitioner's Handbook*. Springe Verlag. 2006. ISBN 0-387-25543-5.

[88] Nathan C. Myers: *Traits: a new and useful template technique*. The C++ Report, June 1995.

[89] C. Nelson and H.-J. Boehm: *Sequencing and the concurrency memory model*. ISO SC22 WG21 TR N2052==06-0122.

[90] Mac OS X 10.1 November 2001 Developer Tools CD Release Notes: *Objective-C++*. `http://developer.apple.com/releasenotes/Cocoa/Objective-C++.html`

[91] Leonie V. Rose and Bjarne Stroustrup: *Complex Arithmetic in C++*. Internal AT&T Bell Labs Technical Memorandum. January 1984. Reprinted in AT&T C++ Translator Release Notes. November 1985.

[92] P. Rovner, R. Levin, and J. Wick: *On extending Modula-2 for building large integrated systems*. DEC research report #3. 1985.

[93] J. Schilling: *Optimizing Away C++ Exception Handling* ACM SIGPLAN Notices. August 1998.

[94] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker,

and Jack Dongarra: *MPI: The Complete Reference* The MIT Press. 1995. http://www-unix.mcs.anl.gov/mpi/.

[95] D. C. Schmidt and S. D. Huston: *Network programming using C++*. Addison-Wesley Vol 1, 2001. Vol. 2, 2003. ISBN 0-201-60464-7 and ISBN 0-201-79525-6.

[96] Jeremy G. Siek and Andrew Lumsdaine: *The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra* ISCOPE '98. http://www.osl.iu.edu/research/mtl.

[97] J. Siek et al.: *Concepts for C++*. ISO SC22 WG21 WG21-N1758.

[98] Jeremy G. Siek and Walid Taha: *A Semantic Analysis of C++ Templates*. Proc. ECOOP 2006.

[99] Yannis Smargdakis: *Functional programming with the FC++ library*. ICFP'00.

[100] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban: *AspectC++: an AOP Extension for C++*. Software Developer's Journal. June 2005. http://www.aspectc.org/.

[101] A. A. Stepanov and D. R. Musser: *The Ada Generic Library: Linear List Processing Packages*. Compass Series, Springer-Verlag, 1989.

[102] A. A. Stepanov: *Abstraction Penalty Benchmark, version 1.2 (KAI)*. SGI 1992. Reprinted as Appendix D.3 of [67].

[103] A. Stepanov and M. Lee: *The Standard Template Library*. HP Labs TR HPL-94-34. August 1994.

[104] Alex Stepanov: Foreword to Siek, et al.: *The Boost Graph Library*. Addison-Wesley 2002. ISBN 0-21-72914-8.

[105] Alex Stepanov: personal communications. 2004.

[106] Alex Stepanov: personal communications. 2006.

[107] Christopher Strachey: *Fundamental Concepts in Programming Languages*. Lecture notes for the International Summer School in Computer Programming, Copenhagen, August 1967.

[108] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. Bell Laboratories Computer Science Technical Report CSTR-84. April 1980. Revised, August 1981. Revised yet again and published as [109].

[109] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. ACM SIGPLAN Notices. January 1982. Revised version of [108].

[110] B. Stroustrup: *Data abstraction in C*. Bell Labs Technical Journal. Vol 63. No 8 (Part 2), pp 1701-1732. October 1984.

[111] B. Stroustrup: *A C++ Tutorial*. Proc. 1984 National Communications Forum. September, 1984.

[112] B. Stroustrup: *The C++ Programming Language* ("TC++PL"). Addison-Wesley Longman. Reading, Mass., USA. 1986. ISBN 0-201-12078-X.

[113] Bjarne Stroustrup: *What is Object-Oriented Programming?* Proc. 14th ASU Conference. August 1986. Revised version in Proc. ECOOP'87, May 1987, Springer Verlag Lecture Notes in Computer Science Vol 276. Revised version in *IEEE Software Magazine*. May 1988.

[114] Bjarne Stroustrup: *An overview of C++*. ACM Sigplan Notices, Special Issue. October, 1986

[115] B. Stroustrup and J. E. Shopiro: *A Set of Classes for Co-routine Style Programming*. Proc. USENIX C++ Workshop. November, 1987.

[116] Bjarne Stroustrup: quote from 1988 talk.

[117] Bjarne Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference, Denver, CO. October 1988. Also, USENIX Computer Systems, Vol 2 No 1. Winter 1989.

[118] B. Stroustrup: *The C++ Programming Language*, 2nd Edition ("TC++PL2" or just "TC++PL"). Addison-Wesley Longman. Reading, Mass., USA. 1991. ISBN 0-201-53992-6.

[119] Bjarne Stroustrup and Dmitri Lenkov: *Run-Time Type Identification for C++*. The C++ Report. March 1992. Revised version. Proc. USENIX C++ Conference. Portland, OR. August 1992.

[120] B. Stroustrup: *A History of C++: 1979-1991*. Proc ACM HOPL-II. March 1993. Also in Begin and Gibson (editors): *History of Programming Languages*. Addison-Wesley. 1996. ISBN 0-201-89502-1.

[121] B. Stroustrup: *The Design and Evolution of C++*. ("D&E"). Addison-Wesley Longman. Reading Mass. USA. 1994. ISBN 0-201-54330-3.

[122] B. Stroustrup: *Why C++ is not just an object-oriented programming language*. Proc. ACM OOPSLA 1995.

[123] B. Stroustrup: *Proposal to Acknowledge that Garbage Collection for C++ is Possible*. WG21/N0932 X3J16/96-0114.

[124] B. Stroustrup: *The C++ Programming Language*, 3rd Edition ("TC++PL3" or just "TC++PL"). Addison-Wesley Longman. Reading, Mass., USA. 1997. ISBN 0-201-88954-4.

[125] B. Stroustrup: *Learning Standard C++ as a New Language*. The C/C++ Users Journal. May 1999.

[126] B. Stroustrup: *The C++ Programming Language*, Special Edition ("TC++PL"). Addison-Wesley, February 2000. ISBN 0-201-70073-5.

[127] B. Stroustrup: *C and C++: Siblings*, *C and C++: A Case for Compatibility*, *C and C++: Case Studies in Compatibility*. The C/C++ Users Journal. July, August, and September 2002.

[128] B. Stroustrup: *Why can't I define constraints for my template parameters?*. http://www.research.att.com/~bs/bs_faq2.html#constraints.

[129] B. Stroustrup and G. Dos Reis: *Concepts — Design choices for template argument checking*. ISO SC22 WG21 TR N1522. 2003.

[130] B. Stroustrup: *Concept checking — A more abstract complement to type checking*. ISO SC22 WG21 TR N1510. 2003.

[131] B. Stroustrup: *Abstraction and the C++ machine model*. Proc. ICESS'04. December 2004.

[132] B. Stroustrup, G. Dos Reis: *A concept design* (Rev. 1). ISO

SC22 WG21 TR N1782=05-0042.

[133] B. Stroustrup: *A rationale for semantically enhanced library languages*. ACM LCSD05. October 2005.

[134] B. Stroustrup and G. Dos Reis: *Supporting SELL for High-Performance Computing*. LCPC05. October 2005.

[135] Bjarne Stroustrup and Gabriel Dos Reis: *Initializer lists*. ISO SC22 WG21 TR N1919=05-0179.

[136] B. Stroustrup: *C++ pages*. http://www.research.att.com/~bs/C++.html.

[137] B. Stroustrup: *C++ applications*. http://www.research.att.com/~bs/applications.html.

[138] H. Sutter, D. Miller, and B. Stroustrup: *Strongly Typed Enums* (revison 2). ISO SC22 WG21 TR N2213==07-0073.

[139] H. Sutter and B. Stroustrup: *A name for the null pointer: nullptr* (revision 2). ISO SC22 WG21 TR N1601==04-0041.

[140] Herb Sutter: *A Design Rationale for C++/CLI*, Version 1.1 — February 24, 2006 (later updated with minor editorial fixes). http://www.gotw.ca/publications/C++CLIRationale.pdf.

[141] Numbers supplied by Trolltech. January 2006.

[142] UK C++ panel: *Objection to Fast-track Ballot ECMA-372 in JTC1 N8037*. http://public.research.att.com/~bs/uk-objections.pdf. January 2006.

[143] E. Unruh: *Prime number computation*. ISO SC22 WG21 TR N462==94-0075.

[144] D. Vandevoorde and N. M. Josuttis: *C++ Templates — The Complete Guide*. Addison-Wesley. 2003. ISBN 0-201-73884-2.

[145] D. Vandevoorde: *Right Angle Brackets* (Revision 1). ISO SC22 WG21 TR N1757==05-0017 .

[146] D. Vandevoorde: *Modules in C++* (Version 3). ISO SC22 WG21 TR N1964==06-0034.

[147] Todd Veldhuizen: *expression templates*. C++ Report Magazine, Vol 7 No. 4, May 1995.

[148] Todd Veldhuizen: *Template metaprogramming*. The C++ Report, Vol. 7 No. 5. June 1995.

[149] Todd Veldhuizen: *Arrays in Blitz++*. Proceedings of the 2nd International Scientific Computing in Object Oriented Parallel Environments (ISCOPE'98). 1998.

[150] Todd Veldhuizen: *C++ Templates are Turing Complete* 2003.

[151] Todd L. Veldhuizen: *Guaranteed Optimization for Domain-Specific Programming*. Dagstuhl Seminar of Domain-Specific Program Generation. 2003.

[152] Andre Weinand et al.: *ET++ — An Object-Oriented Application Framework in C++*. Proc. OOPSLA'88. September 1988.

[153] Gregory V. Wilson and Paul Lu: *Parallel programming using C++*. Addison-Wesley. 1996.

[154] P.M. Woodward and S.G. Bond: *Algol 68-R Users Guide*. Her Majesty's Stationery Office, London. 1974. ISBN 0-11-771600-6.

[155] *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics, 3rd edition*, ISBN 1-930934-12-2. http://public.kitware.com/VTK.

[156] *FLTK: Fast Light Toolkit*. http://www.fltk.org/.

[157] Lockheed Martin Corporation: *Joint Strike Fighter air vehicle coding standards for the system development and demonstration program*. Document Number 2RDU00001 Rev C. December 2005. http://www.research.att.com/~bs/JSF-AV-rules.pdf.

[158] Microsoft Foundation Classes.

[159] *Smartwin++: An Open Source C++ GUI library*. http://smartwin.sourceforge.net/.

[160] *WTL: Windows Template Library —- A C++ library for developing Windows applications and UI components*. http://wtl.sourceforge.net.

[161] *gtkmm: C++ Interfaces for GTK+ and GNOME*. http://www.gtkmm.org/

[162] *wxWidgets: A cross platform GUI library*. http://wxwidgets.org/.

[163] Windows threads: *Processes and Threads*. http://msdn.microsoft.com/library/en-us/dllproc/base/processes_and_threads.asp.

[164] Wikipedia: *Java (Sun) — Early history*. http://en.wikipedia.org/wiki/Java_(Sun).