A history of C++

Conference Paper *in* ACM SIGPLAN Notices · March 1993

DOI: 10.1145/154766.155375

CITATIONS
8

READS
3,903

1 author:

Bjarne Stroustrup
Morgan Stanley
157 PUBLICATIONS **10,604** CITATIONS

A History of C++: 1979–1991

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper outlines the history of the C++ programming language. The emphasis is on the ideas, constraints, and people that shaped the language, rather than the minutiae of language features. Key design decisions relating to language features are discussed, but the focus is on the overall design goals and practical constraints. The evolution of C++ is traced from C with Classes to the current ANSI and ISO standards work and the explosion of use, interest, commercial activity, compilers, tools, environments, and libraries.

1 Introduction

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 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.

This paper is organized in roughly chronological order:

§2 *C with Classes: 1979–1983*. This section describes the fundamental design decisions for C++ as they were made for C++'s immediate predecessor.

§3 *From C with Classes to C++: 1982–1985*. This section describes how C++ evolved from C with Classes up until the first commercial release and the printing of the book that defined C++ in October 1985.

§4 *Release 2.0: 1985–1988*. This section describes how C++ evolved during the early years of commercial availability.

§5 *The Explosion in Interest and Use: 1987–*. This section deals with non-language factors, such as the growth of a C++ tools and library industry. It also tries to estimate the impact of commercial competition on the development of C++.

§6 *Standardization: 1988–*. This section describes the way C++ continues to evolve under the pressures of heavy use in diverse application areas, and how the C++ community handles this challenge through formal ISO and ANSI standardization.

§7 *Retrospective.* This section considers how C++ met its design goals, how it might have been a better language, and how it might become an even more useful tool.

Most effort have been expended on the early years because the design decisions taken early determined the further development of the language. It is also easier to maintain a historical

perspective when one has had many years to observe the consequences of decisions.

Essential language features are presented to make this paper approachable by a non-C++ specialist. However, the emphasis in on the people, ideas, and constraints that shaped C++ rather than on detailed descriptions of those language features or their use. For a description of what C++ is today and how to use it see [Stroustrup,1991]†.

2 C with Classes

C++ evolved from an earlier version called C with Classes. The work and experience with C with Classes from 1979 to 1983 determined the shape of C++.

2.1 Prehistory

The prehistory of C++ – the couple of years before the idea of adding Simula-like features to C occurred to me – is important because during this time the criteria and ideals that later shaped C++ emerged. I was working on my Ph.D. thesis in the Computing Laboratory of Cambridge University in England. My aim was to study alternatives for the organization of system software for a distributed system. The conceptual framework was provided by the capability-based Cambridge CAP computer and its experimental and continuously evolving operating system. The details of this work and its outcome [Stroustrup,1979a] are of little relevance to C++. What is relevant, though, was the focus on composing software out of well-delimited modules and that the main experimental tool was a relatively large and detailed simulator I wrote for simulating software running on a distributed system.

The initial version of this simulator was written in Simula and ran on the Cambridge University computer center's IBM 360/165 mainframe. It was a pleasure to write that simulator. The features of Simula were almost ideal for the purpose and I was particularly impressed by the way the concepts of the language helped me think about the problems in my application. The class concept allowed me to map my application concepts into the language constructs in a direct way that made my code more readable than I had seen in any other language. The way Simula classes can act as co-routines made the inherent concurrency of my application easy to express. For example, an object of class `computer` could trivially be made to work in pseudo-parallel with other objects of class `computer`. Class hierarchies were used to express variants of application level concepts. For example, different types of computers could be expressed as classes derived from class `computer` and different types of inter-module communication mechanisms could be expressed as classes derived from class `IPC`. The use of class hierarchies was not heavy, though; the use of classes to express concurrency was much more important in the organization of my simulator.

During writing and initial debugging I acquired a great respect for the expressiveness of Simula's type system and the ability of its compiler's ability to catch type errors. The observation was that a type error almost invariably reflected either a silly programming error or a conceptual flaw in the design. The latter was by far the most significant and a help that I had not experienced in the use of more primitive ''strong'' type systems. In contrast, I had found Pascal's type system worse than useless – a strait jacket that caused more problems than it solved by forcing me to warp my designs to suit an implementation-oriented artifact. The perceived contrast between the rigidity of Pascal and the flexibility of Simula was essential for the development of C++. Simula's class concept was seen as the key difference and ever since I have seen classes as the proper primary focus of program design.

---

† Author's note: It is now 1995 – 4 years since I first completed this paper. Rather than rewriting major sections of this paper to for the final book, I have added footnotes where needed to reflect recent events. Most of the themes of this paper are explored further in [Stroustrup,1994]. So are many issues related to the design and evolution that couldn't be included here.

I had used Simula before (during my studies at the University of Aarhus, Denmark), but was very pleasantly surprised by the way the mechanisms of the Simula language became increasingly helpful as the size of the program increased. The class and co-routine mechanisms of Simula and the comprehensive type checking mechanisms ensured that problems and errors did not (as I – and I guess most people – would have expected) grow linearly or more than linearly with the size of the program. Instead, the total program acted more like a collection of small (and therefore easy to write, comprehend, and debug) programs rather than a single large program.

The implementation of Simula, however, did not scale in the same way and as a result the whole project came close to disaster. My conclusion at the time was that the Simula implementation (as opposed to the Simula language) was geared to relatively small programs and was inherently unsuited for larger programs [Stroustrup,1979a]. Link times for separately compiled classes were abysmal: It took longer to compile 1/30th of the program and link it to a precompiled version of the rest than it took to compile and link the program as a monolith. This I believe to be more a problem with the mainframe linker than with Simula, but it was still a burden. On top of that, the run-time performance was such that there was no hope of using the simulator to obtain real data. The poor run-time characteristics were a function of the language and its implementation rather than a function of the application. The overhead problems were fundamental to Simula and could not be remedied. The cost arose from several language features and their interactions: run-time type checking, guaranteed initialization of variables, concurrency support, and garbage collection of both user-allocated objects and procedure activation records. For example, measurements showed that more than 80% of the time was spent in the garbage collector despite the fact that resource management was part of the the simulated system so that no garbage was ever produced. Simula implementations are better these days (15 years later), but the order-of-magnitude improvement relative to systems programming languages still has not (to the best of my knowledge) materialized.

To avoid terminating the project I re-wrote the simulator in BCPL and ran it on the experimental CAP computer. The experience of coding and debugging the simulator in BCPL was horrible. BCPL makes C look like a very high level language and provides absolutely no type checking or run-time support. The resulting simulator did, however, run suitably fast and gave a whole range of useful results that clarified many issues for me and provided the basis for several papers on operating system issues [Stroustrup,1978] [Stroustrup,1979b] [Stroustrup,1981a].

Upon leaving Cambridge, I swore never again to attack a problem with tools as unsuitable as those I had suffered while designing and implementing the simulator. The significance of this to C++ was the notion I had evolved of what constituted a ‘‘suitable tool’’ for projects such as the writing of a significant simulator, an operating system, and similar systems programming tasks:

[1] 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 strong (that is, static) checking of a type system based on classes. This I saw as support for the process of inventing programs, as support for design rather than just support for implementation.

[2] A good tool would produce programs that ran as fast as BCPL programs and share BCPL’s ability to easily 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 get caught by inherent limitations in a single language.

[3] A good tool should also allow for 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.

Not all of these criteria were fully formed when I left Cambridge, but several were and more matured on further reflection on my experience with the simulator, on programs written over the next couple of years, and on the experiences of others as learned through discussions and reading of code. C++ as defined at the time of release 2.0 strictly fulfills these criteria; the fundamental tensions in the effort to design templates and exception handling mechanisms for C++ arise from the need to depart from some aspects of these criteria. I think the most important aspect of these criteria is that they are only loosely connected with specific programming language features. Rather, they specify constraints on a solution.

My background in operating systems work and my interest in modularization and communication had permanent effects on C++. The C++ model of protection, for example, is based on the notion of granting and transferring access rights, the distinction between initialization and assignment has its root in thoughts about transferring capabilities, and the design of C++'s exception handling mechanism was influenced by work on fault tolerant systems done by Brian Randell's group in Newcastle in the seventies.

2.2 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, where I have worked ever since. Two sub-problems 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 a pre-processor, called Cpre, that added Simula-like classes to C running and in 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, the task system supporting a co-routine style of programming [Stroustrup,1980b] [Stroustrup,1987b] [Shopiro,1987], was crucial to the usefulness of ''C with Classes,'' as the language accepted by the pre-processor was called, in these 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. This decision has been reconfirmed repeatedly by me and my colleagues, by other C++ users, and by the C++ standards committee. There are many applications for which support for concurrency is essential, but there is no one dominant model for concurrency support; thus when support is needed it should be provided through a library or a special purpose extension so that a particular form of concurrency support does not preclude other forms.

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.

An early description of C with Classes was published as a Bell Labs technical report in April 1980 [Stroustrup,1980a] and later in SIGPLAN Notices. The SIGPLAN paper was in April 1982 followed by a more detailed Bell Labs technical report *Adding Classes to the C Language: An Exercise in Language Evolution* [Stroustrup,1982] that was later published in *Software: Practice and Experience*. These papers set a good example by describing only features that were fully implemented and had been used. This was in accordance with a long standing tradition of Bell Labs Computing Science Research Center; that policy has been modified only where more openness about the future of C++ became needed to ensure a free and open debate over the evolution of C++ among its many non-AT&T users.

C with Classes was explicitly designed to allow better organization of programs; ''computation'' was considered a problem solved by C. I was very concerned that improved program structure was not achieved at the expense of run-time overheads compared to C. The explicit aim was to match C in terms of run-time, code compactness, and data compactness. To wit: Someone once demonstrated a 3% systematic decrease in overall run-time efficiency compared with C. This was considered unacceptable and the overhead promptly removed. Similarly, to ensure layout compatibility with C and thereby avoid space overheads, no ''house-keeping data'' was placed in class objects.

Another major concern was to avoid restrictions on the domain where C with Classes could be used. The ideal – which was achieved – was that C with Classes could be used for whatever C could be used for. This implied that in addition to matching C in efficiency, C with Classes could not provide benefits at the expense of removing ''dangerous'' or ''ugly'' features of C. This observation/principle had to be repeated often to people (rarely C with Classes users) who wanted C with Classes made safer by increasing static type checking along the lines of early Pascal. The alternative way of providing ''safety,'' inserting run-time checks for all unsafe operations, was (and is) considered reasonable for debugging environments but the language could not guarantee such checks without leaving C with a large advantage in run-time and space efficiency. Consequently, such checks were not provided for C with Classes, though C++ environments exist that provide such checks for debugging. In addition, users can and do insert run-time checks (assertions [Stroustrup,1991]) where needed and affordable.

C allows quite low-level operations such as bit manipulation and choosing between different sizes of integers. There are also facilities, such as explicit unchecked type conversions, for deliberately breaking the type system. C with Classes and later C++ follow this path by retaining the low-level and unsafe features of C. In contrast to C, C++ systematically eliminates the need to use such features except where they are essential and performs unsafe operations only at the explicit request of the programmer. I strongly felt then, as I still do, that there is no one right way of writing every program and a language designer has no business of trying to *force* programmers to use a particular style. The language designer does, on the other hand, have an obligation to encourage and support a variety of styles and practices that have proven effective and to provide language features and tools to help programmers avoid the well known traps and pitfalls.

2.3 Feature overview

The features provided in the initial 1980 implementation can be summarized:

[1] classes,
[2] derived classes,
[3] public/private access control,
[4] constructors and destructors,
[5] call and return functions (§2.4.8),
[6] `friend` classes,

[7] type checking and conversion of function arguments

During 1981 three more features were added:

[8] inline functions,

[9] default arguments,

[10] overloading of the assignment operator

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. Having C as a subset dramatically reduced the support and documentation work needed. This was most important because for several years I did all of the C with Classes and later C++ documentation and support in addition to doing the experimentation, design, and implementation. Having all C features available further ensured that no limitations introduced through prejudice or lack of foresight on my part would deprive a user of features already available in C. Naturally, portability to machines supporting C was ensured. Initially, C with Classes was implemented and used on a DEC PDP/11 but soon it was ported to machines such as DEC VAX and Motorola 68000 based machines.

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'' [Stroustrup,1980a]. Support for object-oriented programming was not claimed until the provision of virtual functions in C++ in 1983 [Stroustrup,1984a].

2.4 Feature Details

Clearly, the most important aspect of C with Classes – and later of C++ – was the class concept. Many aspects of the C with Classes class concept can be observed by examining a simple example from [Stroustrup,1980a]:

```
class stack {
    char    s[SIZE];  /* array of characters */
    char *  min;      /* pointer to bottom of stack */
    char *  top;      /* pointer to top of stack */
    char *  max;      /* pointer to top of allocated space */
    void    new();    /* initialization function (constructor) */
public:
    void    push(char);
    char    pop();
};
```

A class is a user-defined data type. A class specifies the type of the class members that define the representation of a variable of the type (an object of the class), specifies the set of operations (functions) that manipulate such objects, and specifies the access users have to these members. Member functions are typically defined ''elsewhere:''

```
char stack.pop()
{
    if (top <= min) error("stack underflow");
    return *(--top);
}
```

Objects of class `stack` can now be defined and used:

```
class stack s1, s2;           /* two variables of class 'stack' */
class stack * p1 = &s2;       /* 'p1' points to 's2' */
class stack * p2 = new stack; /* 'p2' points to stack object
                                 allocated on free store */

s1.push('h');   /* use object directly */
p1->push('s');  /* use object through pointer */
```

Several key design decisions are reflected here:

[1] C with Classes follows Simula in letting the programmer specify types from which variables (objects) can be created, rather than, say, the Modula approach of specifying a module as a collection of objects and functions. In C with Classes (as in C++), a class is a type. This is a key notion in C++. When `class` means user-defined type in C++ why didn't I call it `type`? I chose `class` primarily because I dislike inventing new terminology and found Simula's quite adequate in most cases.

[2] The representation of objects of the user-defined type is part of the class declaration. This has far-reaching implications. For example, it means that true local variables can be implemented without the use of free store (heap store, dynamic store) or garbage collection. It also means that a function must be recompiled the representation of an object it uses directly is changed. See §4.3 for C++ facilities for expressing interfaces that avoid such recompilation.

[3] Compile time access control is used to restrict access to the representation. By default, only the functions mentioned in the class declaration can use names of class members. Members (usually function members) specified in the public interface – the declarations after the `public:` label – can be used by other code.

[4] The full type (including both the return type and the argument types) of a function is specified for function members. Static (compile-time) type checking is based on this type specification. This differed from C at the time, where function argument types were neither specified in interfaces nor checked in calls.

[5] Function definitions are typically specified ''elsewhere'' to make a class more like an interface specification than a lexical mechanism for organizing source code. This implies that separate compilation for class member functions and their users is easy and the linker technology traditionally used for C is sufficient to support C++.

[6] The function `new()` is a constructor, a function with a special meaning to the compiler. Such functions provided guarantees about classes. In this case, the guarantee is that the constructor – known somewhat confusingly as a `new`-function at the time – is guaranteed to be called to initialize every object of its class before the first use of the object.

[7] Both pointers and non-pointer types are provided (as in both C and Simula).

Much of the further development of C with Classes and C++ can be seen as exploring the consequences of these design choices, exploiting their good sides, and compensating for the problems caused by their bad sides. Many, but by no means all, of the implications of these design choices were understood at the time; [Stroustrup,1980a] is dated April 3, 1980. This section tries to explain what was understood at the time and give pointers to sections explaining later consequences and realizations.

2.4.1 Run-time Efficiency

In Simula, it is not possible to have local or global variables of class types; that is, every object of a class must be allocated on the free store using the `new` operator. Measurements of my Cambridge simulator had convinced me that this was a major source of inefficiency. Later, Karel Babcisky from the Norwegian Computer Centre presented data on Simula run-time performance that confirmed my conjecture [Babcisky,1984]. For that reason alone, I wanted global and local

variables of class types.

In addition, having different rules for the creation and scope of built-in and user-defined types is inelegant and I felt that on occasion my programming style had been cramped by absence of local and global class variables in Simula. Similarly, I had on occasion missed the ability to have pointers to built-in types in Simula so I wanted the C notion of pointers to apply uniformly over user-defined and built-in types. This is the origin of the notion that over the years grew into a ''principle'' for C++: User-defined and built-in types should behave the same relative to the language rules and receive the same degree of support from the language and its associated tools. When the ideal was formulated built-in types received by far the best support, but C++ has overshot that target so that built-in types now receive slightly inferior support compared to user-defined types.

The initial version of C with Classes did not provide inline functions to take further advantage of the availability of the representation. Inline functions were soon provided, though. The general reason for the introduction of inline functions was worry that the cost of crossing a protection barrier would cause people to refrain from using classes to hide representation. In particular, [Stroustrup,1982] observes that people had made data members public to avoid the function call overhead incurred by a constructor for simple classes where only one or two assignments are needed for initialization. The immediate cause for the inclusion of inline functions into C with Classes was a project that couldn't afford function call overhead for some classes involved in real-time processing.

Over the years considerations along these lines grew into the C++ ''principle'' that it was not sufficient to provide a feature, it had to be provided in an affordable form. Most definitely, ''affordable'' was seen as meaning ''affordable on hardware common among developers'' as opposed to ''affordable to researchers with high-end equipment'' or ''affordable in a couple of years when hardware will be cheaper.'' C with Classes was always considered as something to be used *now* or *next month* rather than simply a research project to deliver something for a couple of years hence.

Inlining was considered important for the utility of classes and therefore the issue was more *how* to provide it than *whether* to provide it. Two arguments won the day for the notion of having the programmer select which functions the compiler should try to inline. Firstly, I had poor experiences with languages that left the job of inlining to compilers ''because clearly the compiler knows best.'' The compiler only knows best if it has been programmed to inline and it has a notion of time/space optimization that agrees with mine. My experience with other languages was that only ''the next release'' would actually inline and it would do so according to an internal logic that a programmer couldn't effectively control. To make matters worse C (and therefore C with Classes and later C++) has genuine separate compilation so that a compiler never has access to more than a small part of the program (§2.4.2). Inlining a function for which you don't know the source appears feasible given advanced linker and optimizer technology, but such technology wasn't available at the time (and still isn't in most environments). Furthermore, extensive global analysis and optimization easily become unaffordable for large systems – where optimizations are most critical. C with Classes was designed to deliver efficient code given a simple portable implementation on traditional systems. Given that, the programmer had to help. Even today, the choice seems right.

2.4.2 The Linkage Model

The issue of how separately compiled programs are linked together is critical for any programming language and to some extent determines the features the language can provide. One of the critical influences on the development of C with Classes and C++ was the decision that

[1] Separate compilation should be possible with traditional C/Fortran UNIX/DOS style linkers.

[2] Linkage should in principle be type safe.
[3] Linkage should not require any form of database (though one could be used to improve a given implementation).
[4] Linkage to program fragments written in other languages such as C, assembler, and Fortran should be easy and efficient.

C uses ''header files'' to ensure consistent separate compilation. Declarations of data structure layouts, functions, variables, and constants are placed in header files that are typically textually included into every source file that needs the declarations. Consistency is ensured by placing adequate information in the header files and ensuring that the header files are consistently included. C++ follows this model up to a point.

The reason that layout information can be present in a C++ class declaration (though it doesn't *have* to be; see §4.3) is to ensure that the declaration and use of true local variables is easy and efficient. For example:

```
void f()
{
        stack s;
        int c;
        s.push('h');
        c = s.pop();
}
```

Using the `stack` declaration from §2.4, even a simple-minded C with Classes implementation can ensure that no use is made of free store for this example, that the call of `pop()` is inlined so that no function call overhead is incurred and that the non-inlined call of `push()` can invoke a separately compiled function `pop()`. In this, C++ resembles Ada [Ichbiah,1979].

At the time, I felt that there was a tradeoff between having separate interface and implementation declarations (as in Modula2) plus a tool (linker) for matching them up, and having a single class declaration plus a tool (a dependency analyzer) that considered the interface part separately from the implementation details for the purposes of re-compilation. It appears that I underestimated the complexity of the latter and also that the proponents of the former approach underestimate the cost (in terms of porting problems and run time overheads) of the former.

The concern for simple-minded implementations was partly a necessity caused by the lack of resources for developing C with Classes and partly a distrust of languages and mechanisms that required ''clever'' techniques. An early formulation of a design goal was that C with Classes ''should be implementable without using an algorithm more complicated than a linear search.'' Wherever that rule of thumb was violated – as in the case of function overloading (§3.3.3) – it led to semantics that were more complicated than anyone felt comfortable with and typically also to implementation complications.

The aim – based on my Simula experience – was to design a language that would be easy enough to understand to attract users and easy enough to implement to attract implementers. Only if a relatively simple implementation could be used by a relatively novice user in a relatively unsupportive programming environment to deliver code that compared favorably with C code in development time, correctness, run-time speed, and code size could C with Classes and later C++ expect to survive in competition with C.

This was part of a philosophy of fostering self-sufficiency among users. The aim was always – and explicitly – to develop local expertise in all aspects of using C++. Most organizations must follow the exact opposite strategy. They keep users dependent on services that generates revenue for a central support organization and/or consultants. In my opinion, this contrast is a deep reason for some of the differences between C++ and many other languages.

The decision to work in the relatively primitive – and almost universally available – framework of the C linking facilities caused the fundamental problem that a C++ compiler must always

work with only partial information about a program. An assumption made about a program could possibly be violated by a program written tomorrow in some other language (such as C, Fortran, or assembler) and linked in – possibly after the program has started executing. This problem surfaces in many contexts. It is hard for an implementation to guarantee

[1] that something is unique,

[2] that (type) information is consistent,

[3] that something is initialized.

In addition, C provides only the feeblest support for the notion of separate name spaces so that avoiding name space pollution by separately written program segments becomes a problem. Over the years, C++ has tried to face all of these challenges without departing from the fundamental model and technology that gives portability, but in the C with Classes days we simply relied on the C technique of header files.

Through the acceptance of the C linker came another ''principle'' for the development of C++: C++ is just one language in a system and not a complete system. In other words, C++ accepts the role of a traditional programming language with a fundamental distinction between the language, the operating system, and other important parts of the programmer's world. This delimits the role of the language in a way that is hard to do for a language, such as Smalltalk or Lisp, that is conceived as a complete system or environment. It makes it essential that a C++ program fragment can call program fragments written in other languages and that a C++ program fragment can itself be called by program fragments written in other languages. Being ''just a language'' also allows C++ implementations to benefit directly from tools written for other languages.

The need for a programming language and the code written in it to be just a cog in a much larger machine is of utmost importance to most industrial users yet such co-existence with other languages and systems was apparently not a major concern to most theoreticians, would-be perfectionists, and academic users. I believe this to be one of the main reasons for C++'s success.

2.4.3 Static Type Checking

I have no recollection of discussions, no design notes, and no recollection of any implementation problems about the introduction of static (''strong'') type checking into C with Classes. The C with Classes syntax and rules, the ones subsequently adopted for the ANSI C standard, simply appeared fully formed in the first C with Classes implementation. After that, a minor series of experiments led to the current (stricter) C++ rules. Static type checking was to me, after my experience with Simula and Algol68, a simple *must* and the only question was exactly how it was to be added.

To avoid breaking C code, it was decided to allow the call of an undeclared function and not perform type checking on such undeclared functions. This was of course a major hole in the type system and several attempts were made to decrease its importance as the major source of programming errors before finally – in C++ – the hole was closed by making a call of an undeclared function illegal. One simple observation defeated all attempts to compromise, and thus maintain a greater degree of C compatibility: As programmers learned C with Classes they lost the ability to find run-time errors caused by simple type errors. Having come to rely on the type checking and type conversion provided by C with Classes or C++, they lost the ability to quickly find the ''silly errors'' that creep into C programs through the lack of checking. Further, they failed to take the precautions against such silly errors that good C programmers take as a matter of course. After all, ''such errors don't happen in C with Classes.'' Thus, as the frequency of run-time errors caused by uncaught argument type errors goes down their seriousness and the time needed to find them goes up. The result was seriously annoyed programmers demanding further tightening of the type system.

The most interesting experiment with ''incomplete static checking'' was the technique of allowing calls of undeclared functions, but noting the type of the arguments used so that a

consistency check could be done when further calls were seen. When Walter Bright many years later independently discovered this trick he named it ‘‘autoprototyping,’’ using the ANSI C term *prototype* for a function declaration. The experience was that autoprototyping caught many errors and initially increased a programmer’s confidence in the type system. However, since consistent errors and errors in a function called only once in a compilation were not caught, autoprototyping ultimately destroyed programmer confidence in the type checker and induced a sense of paranoia even worse than I have seen in C or BCPL programmers.

C with Classes introduced the notation `f(void)` for a function `f` that takes no arguments as a contrast to `f()` that in C declares a function that can take any number of arguments of any type without any type check. My users soon convinced me, however, that the `f(void)` notation wasn’t very elegant, and that having functions declared `f()` accept arguments wasn’t very intuitive. Consequently, the result of the experiment was to have `f()` mean a function `f` that takes no arguments, as any novice would expect. It took support from both Doug McIlroy and Dennis Ritchie for me to build up courage to make this break from C. Only after they used the word *abomination* about `f(void)` did I dare give `f()` the obvious meaning. However, to this day C’s type rules are much laxer than C++’s and any use of `f()` as a function declaration is incompatible between the two languages.

Another early attempt to tighten C with Classes’ type rules was to disallow ‘‘information destroying’’ implicit conversions. Like others, I had been badly bitten by implicit `long` to `int` and `int` to `char` conversions. I decided to try to ban all implicit conversions that were not value preserving; that is, to require an explicit conversion operator wherever a larger object was stored into a smaller. The experiment failed miserably. Every C program I looked at contained large numbers of assignments of `int`s to `char` variables. Naturally, since these were working programs, most of these assignments were perfectly safe. That is, either the value was small enough not to become truncated or the truncation was expected or at least harmless in that particular context. There was no willingness in the C with Classes community to make such a break from C. I’m still looking for ways to compensate for these problems.

2.4.4 Why C?

A common question at C with Classes presentations was ‘‘Why use C? Why didn’t you build on, say, Pascal?’’ One version of my answer can be found in [Stroustrup,1986b]:

‘‘C is clearly not the cleanest language ever designed nor the easiest to use so why do so many people use it?

[1] 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.

[2] 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.

[3] 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.

[4] 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. Designing ‘‘a better C’’ implies compensating for the major problems involved in writing, debugging, and maintaining C programs *without compromising the advantages of C*. C++ preserves all these advantages and compatibility with C at the cost of abandoning claims to perfection and of some compiler and language complexity. However, designing a language ‘‘from scratch’’ does not ensure perfection and the C++ compilers compare favorably in run-time, have better error detection and reporting, and equal the C compilers in code quality.’’

This formulation is more polished than I could have managed in the early C with Classes days, but it does capture the essence of what I considered important about C and that I did not want to lose in C with Classes. Pascal was considered a toy language [Kernighan,1981], 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 manageable.

At the time, I considered Modula-2, Ada, Smalltalk, Mesa, and Clu as alternatives to C and as sources for ideas for C++ [Stroustrup,1984b] so there was no shortage of inspiration. However, only C, Simula, Algol68, and in one case BCPL left noticeable traces in C++ as released in 1985. Simula gave classes, Algol68 operator overloading (§3.3.3), references (§3.3.4), and the ability to declare variables anywhere in a block (§3.3.1), and BCPL gave // comments (§3.3.1).

There were several reasons for avoiding major departures from C style. I saw the merging of C’s strengths as a systems programming language with Simula’s strengths for organizing programs as a significant challenge in itself. Adding significant features from other languages could easily lead to a ‘‘shopping list’’ language and destroy the integrity of the resulting language. To quote from [Stroustrup,1986b]:

> ‘‘A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed and a set of concepts for the programmer to use when thinking about what can be done. The first aspect ideally requires a language that is ‘‘close to the machine’’, so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The C language was primarily designed with this in mind. The second aspect ideally requires a language that is ‘‘close to the problem to be solved’’ so that the concepts of a solution can be expressed directly and concisely. The facilities added to C to create C++ were primarily designed with this in mind.’’

Again this formulation is more polished than I could have managed during the early stages of the design of C with Classes, but the general idea was clear. Departures from the known and proven techniques of C and Simula would have to wait for further experience with C with Classes and C++ and further experiments. I firmly believe – and believed then – that language design is not just design from first principles but also an art that requires experience, experiments, and sound engineering tradeoffs. Adding a major feature or concept to a language should not be a leap of faith but a deliberate action based on experience and fitting into a framework of other features and ideas of how the resulting language can be used. The post-1985 evolution of C++ shows the influence of ideas from Ada, Clu, and ML.

2.4.5 Syntax Problems

Could I have ‘‘fixed’’ the most annoying deficiencies of the C syntax and semantics at some point before C++ was made generally available? Could I have done so without removing useful features (to C with Classes’ users in their environments – as opposed to an ideal world) or introducing incompatibilities that were unacceptable to C programmers wanting to migrate to C with Classes? I think not. In some cases, I tried, but I backed out my changes after complaints from

outraged users. The part of the C syntax I disliked most was the the declaration syntax. Having both prefix and postfix declarator operators cause a fair amount of confusion. So does allowing the type specifier to be left out (meaning `int` by default)†.

My eventual rationale for leaving things as they were was that any new syntax would (temporarily at least) add complexity to a known mess. Also, even though the old style is a boon to teachers of trivia and to people wanting to ridicule C, it is not a significant problem for C programmers. In this case, I'm not sure if I did the right thing, though. The agony to me and other C++ implementers, documenters, and tool builders caused by the perversities of syntax has been significant. Users can – and do – of course insulate themselves from such problems by writing in a small and easily understood subset of the C/C++ declaration syntax.

A significant syntactic simplification for the benefit of users was introduced into C++ at the cost of some extra work to implementers and some C compatibility problems. In C, the name of a structure, a ''structure tag,'' must always be preceded by the keyword `struct`. For example

```
struct buffer a;    /* 'struct' is necessary in C */
```

In the context of C with Classes, this had annoyed me for some time because it made user-defined types second class citizens syntactically. Given my lack of success with other attempts to clean up the syntax, I was reluctant and only made the change – at the time where C with Classes was mutated into C++ – at the urging of Tom Cargill. The name of a `struct` or a `class` is now a type name and requires no special syntactic identification:

```
buffer a;        // C++
```

The resulting fights over C compatibility lasted for years (see also §3.4).

2.4.6 Derived Classes

The derived class concept is C++'s version of Simula's prefixed class notion and thus a sibling of Smalltalk's subclass concept. The names *derived* class and *base* class were chosen because I never could remember what was *sub* and what was *super* and observed that I was not the only one with this particular problem. It was also noted that many people found it counterintuitive that a subclass typically has *more* information than its superclass. In inventing the terms derived class and base class, I departed from my usual principle of not inventing new names where old ones exist. In my defense, I note that I have never observed any confusion about what is base and what is derived among C++ programmers and that the terms are trivially easy to learn even for people without a grounding in mathematics.

The C with Classes concept was provided without any form of run-time support. In particular, the Simula (and C++) concept of a virtual function was missing. The reason for this was that I – with reason I think – doubted my ability to teach people how to use them and even more my ability to convince people that a virtual function is as efficient in time and space as an ordinary function as typically used. Often, people with Simula and Smalltalk experience still don't quite believe that until they have had the C++ implementation explained to them in detail – and many still harbor irrational doubts after that.

Even without virtual functions, derived classes in C with Classes were useful for building new data structures out of old ones and for associating operations with the resulting types. In particular, as explained in [Stroustrup,1980] and [Stroustrup,1982], they allowed list classes to be defined and also task classes.

In the absence of virtual functions, a user could use objects of a derived class and treat base classes as implementation details (only). Alternatively, an explicit type field could be introduced in a base class and used together with explicit type casts. The former strategy was used for tasks

† In 1995, the C++ standards committee finally banned ''implicit `int`'' in declarations.

where the user only sees specific derived task classes and ‘‘the system’’ sees only the task base classes. The latter strategy was used for various application classes where, in effect, a base class was used to implement a variant record for a set of derived classes. Much of the effort in C with Classes and later C++ has been to ensure that programmers needn’t write such code. Most important in my thinking at the time and in my own code was the combination of base classes, explicit type conversions, and (occasionally) macros to provide generic container classes. Eventually, these techniques matured into C++’s template facility and the techniques for using templates together with base classes to express commonality among instantiated templates (§6.3).

2.4.7 The Protection Model

Before starting work on C with Classes, I worked with operating systems. The notions of protection from the Cambridge CAP computer and similar systems – rather than any work in programming languages – inspired the C++ protection mechanisms. The class is the unit of protection and the fundamental rule is that you cannot grant yourself access to a class; only the declarations placed in the class declaration (supposedly by its owner) can grant access. By default, all information is private.

Access is granted by declaring a function in the public part of a class declaration, or by specifying a function or a class as a `friend`. Initially, only classes could be friends, thus granting access to all member functions of the friend class, but later it was found convenient to be able to grant access (friendship) to individual functions. In particular, it was found useful to be able to grant access to global functions.

A friendship declaration was seen as a mechanism similar to that of one protection domain granting a read-write capability to another.

Even in the first version of C with Classes, the protection model applied to base classes as well as members. Thus a class could be either publicly or privately derived from another. The private/public distinction for base classes predates the debate on implementation inheritance vs interface inheritance by about 5 years [Snyder,1986] [Liskov,1987]. If you want to inherit an implementation only, you use private derivation in C++. Public derivation gives users of the derived class access to the interface provided by the base class. Private derivation leaves the base as an implementation detail; even the public members of the private base class are inaccessible except through the interface explicitly provided for the derived class.

To provide ‘‘semi-transparent scopes’’ a mechanism was provided to allow individual public names from a private base class to be made public [Stroustrup,1982].

2.4.8 Run-time Guarantees

The access control mechanisms described above simply prevent unauthorized access. A second kind of guarantee was provided by ‘‘special member functions,’’ such as constructors, that were recognized and implicitly invoked by the compiler. The idea was to allow the programmer to establish guarantees, sometimes called ‘‘invariants,’’ that other member function could rely on. Curiously enough, the initial implementation contained a feature that is not provided by C++ but is often requested. In C with Classes, it was possible to define a function that would implicitly be called before every call of every member function (except the constructor) and another that would be implicitly called before every return from every member function. They were called `call` and `return` functions. They were used to provide synchronization for the monitor class in the original task library [Stroustrup,1980b]:

```
class monitor : object {
        /* ... */
        call() { /* grab lock */ }
        return() { /* release lock */ }
};
```

These are similar in intent to the CLOS `:before` and `:after` methods. Call and return functions were removed from the language because nobody (but me) used them and because I seemed to have completely failed to convince people that `call()` and `return()` had important uses. In 1987 Mike Tiemann suggested an alternative solution called ''wrappers'' [Tiemann,1987], but at the USENIX implementors' workshop in Estes Park this idea was determined to have too many problems to be accepted into C++.

2.4.9 Features Considered, but not Provided

In the early days, many features were considered that later appeared in C++ or are still discussed. These included virtual functions, `static` members, templates, and multiple inheritance. However,

> ''All of these generalizations have their uses, but every ''feature'' of a language takes time and effort to design, implement, document, and learn.'' ... ''The base class concept is an engineering compromise, like the C class concept [Stroustrup,1982].''

I just wish I had explicitly mentioned the need for experience. With that, the case against featurism and for a pragmatic approach would have been complete.

The possibility of automatic garbage collection was considered on several occasions before 1985 and deemed unsuitable for a language already in use for real-time processing and hard-core systems tasks such as device drivers. In those days, garbage collectors were less sophisticated than they are today and the processing power and memory capacity of the average computer were small fractions of what today's systems offer. My personal experience with Simula and reports of other GC-based systems convinced me that GC was unaffordable by me and my colleagues for the kind of applications we were writing. Had C with Classes (or even C++) been defined to require automatic garbage collection it would have been more elegant, but stillborn.

Direct support for concurrency was also considered but rejected in favor of a library based approach (§2.2).

2.5 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 – and still provides – a possibly unique environment for such work. When I joined 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 others have 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 that 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 [Stroustrup,1986b]:

> ''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.''

Only after C++ was an established language did more conventional organizational structures emerge and even then I was officially in charge of the reference manual and had the final say over what went into it until that task was handed over to the ANSI C++ committee in early 1990. On the other hand, after the first few months I never had the freedom to design just for the sake of designing something beautiful or to make arbitrary changes in the language as it stood at any given time. Whatever I considered a language feature required an implementation to make it real, and any change or extension required the concurrence and usually enthusiasm of key C with Classes and later C++ users.

Since there was no guaranteed user population, the language and its implementations could only survive by serving the needs of its users well enough to counteract the organizational pull of established languages and the marketing hype of newer languages.

C with Classes grew through discussions with people in the Computing Science Research Center and early users there and elsewhere in the Labs. Most of C with Classes and later C++ was designed on somebody else's blackboard and the rest on mine. Most such ideas were rejected as being too elaborate, too limited in usefulness, too hard to implement, too hard to teach for use in real projects, not efficient enough in time or space, too incompatible with C, or simply too weird. The few ideas that made it through this filter – invariably involving discussions among at least two people – I then implemented. Typically, the idea mutated through the effort of implementation, testing, and early use by me and one or two others. The resulting version was tried on a larger audience and would often mutate a bit further before finding its way into the ''official'' version of C with Classes as shipped by me. Usually, a tutorial was written somewhere along the way. Writing a tutorial was considered an essential design tool, because if a feature cannot be explained simply the burden of supporting it will be too great. This point was never far from my mind because during the early years I *was* the support organization.

In the early days Sandy Fraser, my department head at the time, was very influential. For example, I believe he was the one to encourage me to break from the Simula style of class definition where the complete function definition is included and adopt the style where function definitions are typically elsewhere thus emphasizing the class declaration's role as an interface. Much of C with Classes was designed to allow simulators to be built that could be used in Sandy Fraser's work in network design. The first real application of C with Classes was such network simulators. Sudhir Agrawal was another early user who influenced the development of C with Classes through his work with network simulations. Jonathan Shopiro provided much feedback of the C with Classes design and implementation based on his simulation of a ''dataflow database machine.''

For more general discussions on programming language issues, as opposed to looking at applications to determine which problems needed to be solved, I turned to Dennis Ritchie, Steve Johnson, and in particular Doug McIlroy. Doug McIlroy's influence on the development of both C and C++ cannot be overestimated. I cannot remember a single critical design decision in C++ that I have not discussed at length with Doug. Naturally, we didn't always agree, but I still have a strong reluctance to make a decision that goes against Doug's opinion. He has a knack for being right and an apparently infinite amount of experience and patience.

Since the main design work for C with Classes and C++ was done on blackboards the thinking tended to focus on solutions to ''archetypical'' problems: Small examples that are considered characteristic for a large class of problems. Thus, a good solution to the small example will provide significant help in writing programs dealing with real problems of that class. Many of these problems have entered the C++ literature and folklore through my use of them as examples in my papers, books, and talks. For C with Classes, the example considered most critical was the `task` class that was the basis of the task-library supporting Simula-style simulation. Other key classes

were `queue`, `list`, and `histogram` classes. The `queue` and `list` classes were based on the idea – borrowed from Simula – of providing a `link` class from which users derived their own classes.

The danger inherent in this approach is to create a language and tools that provide elegant solutions to small selected examples yet don't scale to building complete systems or large programs. This was counteracted by the simple fact that C with Classes (and later C++) had to pay for itself during its early years. This ensured that C with Classes couldn't evolve into something that was elegant but useless.

3 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. I defined a medium success as something so useful that it easily paid for itself and its developer, but not so attractive and useful that it would pay for a support and development organization. Thus, continuing with C with Classes and its C pre-processor implementation would condemn me to support C with Classes use indefinitely. I was convinced that there were only two ways out of this dilemma:

[1] Stop supporting C with Classes so that the users would have to go elsewhere (freeing me to do something else).

[2] Develop a new and better language based on my experience with C with Classes that would serve a large enough set of users to pay for a support and development organization (thus freeing me to do something else). At the time I estimated that 5,000 industrial users was the necessary minimum.

The third alternative, increasing the user population through marketing (hype), never occurred to me. What actually happened was that the explosive growth of C++, as the new language was eventually named, kept me so busy that to this day I haven't managed to get sufficiently detached to do something else of significance.

The success of C with Classes was, I think, a simple consequence of meeting its design aim: C with Classes did help organize a large class of programs significantly better than C without the loss of run-time efficiency and without requiring enough cultural changes to make its use infeasible in organizations that were unwilling to undergo major changes. 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. There simply wasn't enough support in C with Classes for people who *were* willing to invest significant efforts 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.

The resulting language was at first still called C with Classes but after a polite request from management it was given the name C84. The reason for the naming was that people had taken to calling C with Classes ''new C,'' and then C. This last abbreviation led to C being called ''plain C,'' ''straight C,'' and ''old C.'' The name C84 was used only for a few months, partly because it was ugly and institutional, partly because there would still be confusion if people dropped the '84.' I asked for ideas for a new name and picked C++ because it was short, had nice interpretations, and wasn't of the form ''adjective C.'' In C, ++ can, depending on context, be read as ''next,'' ''successor,'' or ''increment'' though it is always pronounced ''plus plus.'' The name C++ and its runner up ++C are fertile sources for jokes and puns – almost all of which were known and appreciated before the name was chosen. The name C++ was suggested by Rick Mascitti. It was first used in [Stroustrup,1984b] where it was edited into the final copy in December of 1983.

3.1 Aims

During the 1982–1984 period the aims for C++ gradually became more ambitious and more definite. I had come to see C++ as a language separate from C, and libraries and tools had emerged as areas of work. Because of that, because tools developers within Bell Labs were beginning to show interest in C++, and because I had embarked on a completely new implementation that would become the C++ compiler front-end, Cfront, I had to answer key questions:

[1] Who will the users be?

[2] What kind of systems will they use?

[3] How will I get out of the business of providing tools?

[4] How should the answers to [1], [2], and [3] affect the language definition?

My answer to [1], ''Who will the users be?,'' was that first my friends within Bell Labs and I would use it, then more widespread use within AT&T would provide more experience, then some universities would pick up the ideas and the tools, and finally AT&T and others would be able to make some money by selling the set of tools that had evolved. At some point, the initial and somewhat experimental implementation done by me would be faded out in favor of more ''industrial strength'' implementations by AT&T and others.

This made practical and economic sense; the initial (Cfront) implementation would be tool-poor, portable, and cheap because that was what I, my colleagues, and many university users needed and could afford. Later, there would be ample scope for AT&T and others to provide better tools for more specialized environments. Such better tools aimed primarily at industrial users needn't be cheap either, and would thus be able to pay for the support organizations necessary for large-scale use of the language. That was my answer to [3] ''How will I get out of the business of providing tools?'' Basically, the strategy worked. However, just about every detail actually happened in an unforeseen way.

To get an answer to [2] ''What kind of systems will they use?,'' I simply looked around to see what kind of systems the C with Classes users actually did use. They used everything from boxes that were so small that they couldn't run a compiler to mainframes. They used more operating systems than I had heard of. Consequently, I concluded that extreme portability and the ability to do cross compilation were necessities and that I could make no assumption about the size and speed of the machines running generated code. To build a compiler, however, I would have to make assumptions about the kind of system people would develop their programs on. I assumed that one MIPS plus one Mbyte would be available. That assumption, I considered a bit risky because most of my prospective users at the time had at most part of a PDP11 or some other relatively low-powered and/or timeshared system available.

I did not predict the PC revolution, but by over-shooting my performance target for Cfront I happened to build a compiler that (barely) could run on an IBM PC/AT, thus providing an existence proof that C++ could be an effective language on a PC and thereby spurring commercial software developers to beat it.

As the answer to [4] ''How does all this affect the language definition?'' I concluded that no feature must require really sophisticated compiler or run-time support, that available linkers must be used, and that the code generated would have to be efficient (comparable to C) even initially.

3.2 Cfront

The Cfront compiler front-end for the C84 language was designed and implemented by me between the spring of 1982 and the summer of 1983. The first user outside the computer science research center, Jim Coplien, received his copy in July of 1983. Jim was in a group that had been doing experimental switching work with C with Classes in Bell Labs in Naperville, Illinois for some time.

In that same time period I designed C84, drafted the reference manual published January 1, 1984 [Stroustrup,1984a], designed the `complex` number library and implemented it together

with Leonie Rose [Rose,1984], designed and implemented the first `string` class together with Jonathan Shopiro, maintained and ported the C with Classes implementation, supported the C with Classes users and helped them become C84 users. That was a busy year and a half.

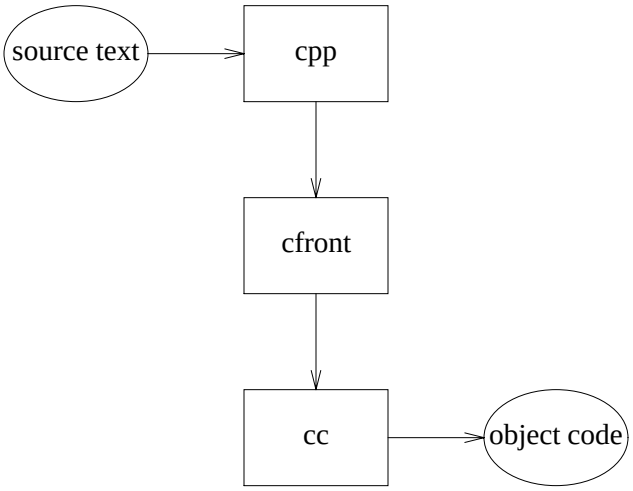
Cfront was (and is) a traditional compiler front-end performing a complete check of the syntax and semantics of the language, building an internal representation of its input, analyzing and rearranging that representation, and finally producing output suitable for some code generator. The internal representation was (is) a graph with one symbol table per scope. The general strategy is to read a source file one global declaration at a time and produce output only when a complete global declaration has been completely analyzed.

The organization of Cfront is fairly traditional except maybe for the use of many symbol tables instead of just one. Cfront was originally written in C with Classes (what else?) and soon transcribed into C84 so that the very first working C++ compiler was done in C++. Even the first version of Cfront used classes heavily, but no virtual functions because they were not available at the start of the project.

The most unusual – for its time – aspect of Cfront was that it generated C code. This has caused no end of confusion. Cfront generated C because I needed extreme portability for an initial implementation and I considered C the most portable assembler around. I could easily have generated some internal back-end format or assembler from Cfront, but that was not what my users needed. No assembler or compiler back-end served more than maybe a quarter of my user community and there was no way that I could produce the, say, six backends needed to serve just 90% of that community. In response to this need, I concluded that using C as a common input format to a large number of code generators was the only reasonable choice. The strategy of building a compiler as a C generator has later become quite popular so that languages such as Ada, CLOS, Eiffel, Modula-3, and Smalltalk have been implemented that way. I got a high degree of portability at a modest cost in compile time overhead. Over the years I have measured this overhead on various systems and found it to be between 25% and 100% of the ‘‘necessary’’ parts of a compilation.

Please note that the C compiler is used as a code generator *only*. Any error message from the C compiler reflects an error in the C compiler or in Cfront, but not in the C++ source text. Every syntactic and semantic error is in principle caught by Cfront, the C++ compiler front-end. I stress this because there has been a long history of confusion about what Cfront was/is. It has been called a preprocessor because it generates C, and for people in the C community (and elsewhere) that has been taken as proof that Cfront was a rather simple program – something like a macro preprocessor. People have thus ‘‘deduced’’ (wrongly) that a line-for-line translation from C++ to C is possible, that symbolic debugging at the C++ level is impossible when Cfront is used, that code generated by Cfront must be inferior to code generated by ‘‘real compilers,’’ that C++ wasn’t a ‘‘real language,’’ etc. Naturally, I have found such unfounded claims most annoying – especially when they were leveled as criticisms of the C++ language. There are now several C++ compilers that use Cfront together with local code generators without going through a C front end. To the user, the only obvious difference is faster compile times.

Cfront is only a compiler front-end and can never be used for real programming by itself. It needs a driver to run the source file through the C preprocessor, Cpp, then run the output of Cpp through Cfront and the output from Cfront through a C compiler:



In addition, the driver must ensure that dynamic (run-time) initialization is done. In Cfront 3.0, the driver becomes yet more elaborate as automatic template instantiation (§6.3) is implemented [McClusky,1992].

As mentioned, I decided to live within the constraints of traditional linkers. However, there was one constraint that I felt was too difficult to live with, yet so silly that I had a chance of fighting it if I had sufficient patience: Most traditional linkers had a very low limit on the number of characters that can be used in external names. A limit of 8 characters was common, and 6 characters and one case only are guaranteed to work as external names in Classical C; ANSI/ISO C accepts that limit also. Given that the name of a member function includes the name of its class and that the type of an overloaded function has to be reflected in the linkage process somehow or other, I had little choice. Consequently I started (in 1982) lobbying for longer names in linkers. I don't know if my efforts actually had any effect, but these days most linkers do give me the much larger number of characters I need. Cfront uses encodings to implement type-safe linkage in a way that makes a limit of 32 characters too low for comfort and even 256 is a bit tight at times (see §3.3.3). In the interim, systems of hash coding of long identifiers have been used with archaic linkers, but that was never completely satisfactory.

Versions of C++ are often named by Cfront release numbers. Release 1.0 was the language as defined in ''*The C++ Programming Language*'' [Stroustrup,1986b].

Releases 1.1 (June 1986) and 1.2 (February 1987) were primarily bug fix releases but also added pointers to members and protected members (§4.1). Release 2.0 was a major cleanup that also introduced multiple inheritance (§4.2) in June 1989. Release 2.1 (April 1990) was primarily a bug fix release that brought Cfront (almost) into line with the definition in the ARM, Ellis & Stroustrup: *The Annotated C++ Reference Manual* [Ellis,1990] (§6.1). Release 3.0 (September 1991) added templates (§6.3) as specified in the ARM. Release 4.0 (due in 1993) is expected to add exception handling (§6.4) as specified in the ARM.

I wrote the first versions of Cfront (1.0, 1.1, 1.2) and maintained them; Steve Dewhurst worked on it with me for a few months before release 1.0 in 1985. Laura Eaves did much of the work on the Cfront parser for release 1.0, 1.1, 2.1, and 3.0. I also did the lion's share of the programming for releases 1.2 and 2.0, but starting with release 1.2, Stan Lippman also spent most of his time on Cfront. George Logothetis, Judy Ward, and Nancy Wilkinson, and Stan Lippman did most of the work for releases 2.1 and 3.0. The work on 2.0 was coordinated by Barbara Moo, and Andrew Koenig organized Cfront testing. Barbara also coordinated releases 1.2, 2.1, and 3.0. Sam Haradhvala from Object Design Inc. did an initial implementation of templates in 1989 that Stan Lippman extended and completed for release 3.0 in 1991. The initial implementation of exception handling in Cfront was done by Hewlett-Packard in 1992. In addition to these people

who have produced code that has found its way into the main version of Cfront, many people have built local C++ compilers from it. Apple, Centerline (formerly Saber), ParcPlace, Sun, HP, and others ship products that contain locally modified versions of Cfront.

3.3 Language Feature Details

The major additions to C with Classes introduced to produce C++ were:

[1] Virtual functions.
[2] Function name and operator overloading.
[3] References.
[4] Constants (`const`).
[5] User-controlled free-store memory control.
[6] Improved type checking

In addition, the notion of call and return functions (§2.4.8) was dropped due to lack of use and many minor details were changed to produce a cleaner language.

3.3.1 Minor Changes

The most visible minor change was the introduction of BCPL-style comments:

```
int a; /* C-style explicitly terminated comment */
int b; // BCPL-style comment terminated by end-of-line
```

Since both styles of comments are allowed people can simply use the style they like best.

The name ''`new`-function'' for constructors had been a source of confusion so the name constructor was introduced.

In C with Classes, a dot was used to express membership of a class as well as expressing selection of a member of a particular object. This had been the cause of some minor confusion and could also be used to construct ambiguous examples. To alleviate this, `::` was introduced to mean membership of class and `.` was retained exclusively for membership of object.

I borrowed the Algol68 notion that a declaration can be introduced wherever it is needed (and not just at the top of some block). Thus, I enabled a ''initialize-only'' or ''single-assignment'' style of programming that is less error-prone than traditional styles. This style is essential for references and constants that cannot be assigned and inherently more efficient for types where default initialization is expensive.

3.3.2 Virtual Functions

The most obvious new feature in C++ – and certainly the one that had the greatest impact on the style of programming one could use for the language – was virtual functions. The idea was borrowed from Simula and presented in a form that was intended to make a simple and efficient implementation easy. The rationale for virtual functions was presented in [Stroustrup,1986b] and [Stroustrup,1986c]. To emphasize the central role of virtual functions in C++ programming I will quote it in detail here [Stroustrup,1986c]:

> ''An abstract data type defines a sort of black box. Once it has been defined, it does not really interact with the rest of the program. There is no way of adapting it to new uses except by modifying its definition. This can lead to severe inflexibility. Consider defining a type `shape` for use in a graphics system. Assume for the moment that the system has to support circles, triangles, and squares. Assume also that you have some classes:
>
> ```
> class point{ /* ... */ };
> class color{ /* ... */ };
> ```
>
> You might define a shape like this:

```
enum kind { circle, triangle, square };

class shape {
    point center;
    color col;
    kind k;
    // representation of shape
public:
    point where()       { return center; }
    void move(point to) { center = to; draw(); }
    void draw();
    void rotate(int);
    // more operations
};
```

The ‘‘type field’’ `k` is necessary to allow operations such as `draw()` and `rotate()` to determine what kind of shape they are dealing with (in a Pascal-like language, one might use a variant record with tag `k`). The function `draw()` might be defined like this:

```
void shape::draw()
{
    switch (k) {
    case circle:
        // draw a circle
        break;
    case triangle:
        // draw a triangle
        break;
    case square:
        // draw a square
    }
}
```

This is a mess. Functions such as `draw()` must ‘‘know about’’ all the kinds of shapes there are. Therefore the code for any such function grows each time a new shape is added to the system. If you define a new shape, every operation on a shape must be examined and (possibly) modified. You are not able to add a new shape to a system unless you have access to the source code for every operation. Since adding a new shape involves ‘‘touching’’ the code of every important operation on shapes, it requires great skill and potentially introduces bugs into the code handling other (older) shapes. The choice of representation of particular shapes can get severely cramped by the requirement that (at least some of) their representation must fit into the typically fixed sized framework presented by the definition of the general type `shape`.

The problem is that there is no distinction between the general properties of any shape (a shape has a color, it can be drawn, etc.) and the properties of a specific shape (a circle is a shape that has a radius, is drawn by a circle-drawing function, etc.). Expressing this distinction and taking advantage of it defines object-oriented programming. A language with constructs that allows this distinction to be expressed and used supports object-oriented programming. Other languages don’t.

The Simula inheritance mechanism provides a solution. First, specify a class that defines the general properties of all shapes:

```
class shape {
    point center;
    color col;
    // ...
public:
    point where() { return center; }
    void move(point to) { center = to; draw(); }
    virtual void draw();
    virtual void rotate(int);
    // ...
};
```

The functions for which the calling interface can be defined, but where the implementation cannot be defined except for a specific shape, have been marked ‘‘virtual’’ (the Simula and C++ term for ‘‘may be re-defined later in a class derived from this one’’). Given this definition, we can write general functions manipulating shapes:

```
void rotate_all(shape** v, int size, int angle)
// rotate all members of vector "v" of size "size" "angle" degrees
{
    for (int i = 0; i < size; i++) v[i].rotate(angle);
}
```

To define a particular shape, we must say that it is a shape and specify its particular properties (including the virtual functions).

```
class circle : public shape {
    int radius;
public:
    void draw() { /* ... */ };
    void rotate(int) {}    // yes, the null function
};
```

In C++, class `circle` is said to be *derived* from class `shape`, and class `shape` is said to be a *base* of class `circle`. An alternative terminology calls `circle` and `shape` subclass and superclass, respectively.’’

For further discussion of virtual functions and object-oriented programming see §3.7 and §4.3.

The key implementation idea was that the set of virtual functions in a class defines a array of pointers to functions so that a call of a virtual function is simply an indirect function call through that array. There is one array per class and one pointer to such an array in each object of a class that has virtual functions.

I don’t remember much interest in virtual functions at the time. Probably I didn’t explain the concepts involved well, but the main reaction I received from people in my immediate vicinity was one of indifference and skepticism. A common opinion was that virtual functions were simply a kind of crippled pointer to function and thus redundant. Worse, it was sometimes argued that a well-designed program wouldn’t need the extensibility and openness provided by virtual functions so that proper analysis would show which non-virtual functions could be called directly. Therefore, the argument went, virtual functions were simply a form of inefficiency. Clearly I disagreed, and added virtual functions anyway.

3.3.3 Overloading

Several people had asked for the ability to overload operators. Operator overloading ‘‘looked neat’’ and I knew from experience with Algol68 how the idea could be made to work. However, I was reluctant to introduce the notion of overloading into C++:

[1] Overloading was reputed to be hard to implement so that compilers would grow to

monstrous size.

[2] Overloading was reputed to be hard to teach and hard to define precisely so that manuals and tutorials would grow to monstrous size.

[3] Code written using operator overloading was reputed to be inherently inefficient.

[4] Overloading was reputed to make code incomprehensible.

If [3] or [4] were true then C++ would be better off without overloading. If [1] or [2] were true then I didn't have the resources to provide overloading.

However, if all of these conjectures were false then overloading would solve some real problems for C++ users. There were people who would like to have complex numbers, matrices, and APL-like vectors in C++. There were people who would like range-checked arrays, multi-dimensional arrays and strings in C++. There were at least two separate applications for which people wanted to overload logical operators such as | (or), & (and), and ^ (exclusive or). The way I saw it, the list was long and would grow with the size and the diversity of the C++ user population. My answer to [4] ''overloading makes code obscure'' was that several of my friends whose opinion I valued and whose experience was measured in decades claimed that their code would become cleaner if they had overloading. So what if one can write obscure code with overloading? It is possible to write obscure code in any language. It matters more how a feature can be used well than how it can be misused.

Next, I convinced myself that overloading wasn't inherently inefficient [Stroustrup,1984c] [Ellis,1990]. The details of the overloading mechanism were mostly worked out on my blackboard and those of Stu Feldman, Doug McIlroy, and Jonathan Shopiro.

Thus, having worked out an answer to [3], ''code written using overloading is efficient,'' I needed to concern myself with [1] and [2], the issue of compiler and language complexity. I first observed that use of classes with overloaded operators, such as `complex` and `string`, was quite easy and didn't put a major burden on the programmer. Next I wrote the manual sections to prove that the added complexity wasn't a serious issue; the manual needed less than a page and a half extra (out of a 42-page manual). Finally, I did the first implementation in two hours using only 18 lines of extra code in Cfront, and I felt I had demonstrated that the fears about definition and implementation complexity were somewhat exaggerated.

Naturally, all these issues were not really tackled in this strict sequential order. However, the emphasis of the work did start with utility issues and slowly drifted to implementation issues. The overloading mechanisms were described in detail in [Stroustrup,1984c] and examples of classes using the mechanisms were written up [Rose,1984] [Shopiro,1985].

In retrospect, I underestimated the complexity of the definition and implementation issues and compounded these problems by trying to isolate overloading mechanisms from the rest of the language semantics. The latter was done out of misguided fear of confusing users. In particular, I required that a declaration

```
overload print;
```

should precede declarations of an overloaded function `print`, such as

```
void print(int);
void print(const char*);
```

I also insisted that ambiguity control should happen in two stages so that resolutions involving built-in operators and conversions would always take precedence over resolutions involving user-defined operations. Maybe this latter was inevitable given the concern for C compatibility and the chaotic nature of the C conversion rules for built-in types. These conversions do *not* constitute a lattice; for example, implicit conversions are allowed both from `int` to `float` and from `float` to `int`. However, the rules for ambiguity resolution were too complicated, caused surprises, and had to be revised for release 2.0. I still consider these rules too complex, but do not see scope for more than minor adjustments.

Requiring explicit `overload` declarations was plain wrong and the requirement was dropped in release 2.0.

3.3.4 References

References were introduced primarily to support operator overloading. C passes every function argument by value, and where passing an object by value would be inefficient or inappropriate the user can pass a pointer. This strategy doesn't work where operator overloading is used. In that case, notational convenience is essential so that a user cannot be expected to insert address-of operators if the objects are large.

Problems with debugging Algol68 convinced me that having references that didn't change what object they referred to after initialization was a good thing. If you wanted to do more complicated pointer manipulation in C++ you can use pointers. Because C++ has both pointers and references it does not need operations for distinguishing operations on the reference itself from operations on the object referred to (like Simula) or the kind of deductive mechanism employed by Algol68.

It is important that `const` references can be initialized by non-lvalues and lvalues of types that require conversion. In particular, this is what allows a Fortran function to be called with a constant:

```
extern "Fortran" float sqrt(const float&); // '&' means reference

sqrt(2); // call by reference
```

Jonathan Shopiro was deeply involved in the discussions that led to the introduction of references. In addition to the obvious uses of references, such as argument, we considered the ability to use references as return types important. This allowed us to have a very simple index operator for a string class:

```
class String {
  // ...
  char& operator[](int index);  // subscript operator
                                // return a reference
};

void f(String& s)
{
  char c1 = ...
  s[i] = c1;      // assign to operator[]'s result
  // ...
  char c2 = s[i]; // assign operator[]'s result
}
```

We considered allowing separate functions for left-hand and right-hand side use of a function but considered using references the simpler alternative even though this implies that we need to introduce additional ''helper classes'' to solve some problems where returning a simple reference isn't enough.

3.3.5 Constants (`const`)

In operating systems, it is common to have access to some piece of memory controlled directly or indirectly by two bits: one that indicates whether a user can write to it and one that indicates whether a user can read it. This idea seemed to me directly applicable to C++ and I considered allowing every type to be specified `readonly` or `writeonly` [Stroustrup,1981b]. The proposal is focused on specifying interfaces rather than on providing symbolic constants for C. Clearly, a `readonly` value is a symbolic constant, but the scope of the proposal is far greater.

Initially, I proposed pointers to `readonly` but not `readonly` pointers. A brief discussion with Dennis Ritchie evolved the idea into the `readonly/writeonly` mechanism that I implemented and proposed to an internal Bell Labs C standards group chaired by Larry Rosler. There, I had my first experience with standards work. I came away from a meeting with an agreement (that is, a vote) that `readonly` would be introduced into C – yes C, not C with Classes or C++ – provided it was renamed `const`. Unfortunately, a vote isn't executable so nothing happened to our C compilers. A while later, the ANSI C committee (X3J11) was formed and the `const` proposal resurfaced there and became part of ANSI/ISO C.

However, in the meantime I had experimented further with `const` in C with Classes and found that `const` was a useful alternative to macros for representing constants only if a global `const`s were implicitly local to their compilation unit. Only in that case could the compiler easily deduce that their value really didn't change and allow simple `const`s in constant evaluations and thus avoid allocating space for such constants and use them in constant expressions. C did not adopt this rule. This makes `const`s far less useful in C than in C++ and leaves C dependent on the preprocessor where C++ programmers can use properly typed and scoped `const`s.

3.3.6 Memory Management

Long before the first C with Classes program was written, I knew that free store (dynamic memory) would be used more heavily in a language with classes than in traditional C programs. This was the reason for the introduction of the `new` and `delete` operators in C with Classes. The `new` operator that both allocates memory from the free store and invokes a constructor to ensure initialization was borrowed from Simula. The `delete` operator was a necessary complement because I did not want C with Classes to depend on a garbage collector. The argument for the `new` operator can be summarized like this. Would you rather write:

```
X* p = new X(2);
```

or

```
struct X * p = (struct X *) malloc(sizeof(struct X));
if (p == 0) error("memory exhausted");
p->init(2);
```

and which version are you most likely to make a mistake in? The arguments against – which were voiced quite a lot at the time – were ''but we don't *really* need it,'' and ''but someone will have used `new` as an identifier.'' Both observations are correct, of course.

Introducing operator `new` thus made the use of free store more convenient and less error-prone. This increased its use even further so that the C free store allocation routine `malloc()` used to implement `new` became the most common performance bottleneck in real systems. This was no real surprise either; the only problem was what to do about it. Having real programs spend 50% or more of their time in `malloc()` wasn't acceptable.

I found per-class allocators and deallocators very effective. The fundamental idea is that free store memory usage is dominated by the allocation and deallocation of lots of small objects from very few classes. Take over the allocation of those objects in a separate allocator and you can save both time and space for those objects and also reduce the amount of fragmentation of the general free store. The mechanism provided for 1.0, ''assignment to `this`,'' was too low level and error-prone and was replaced by a cleaner solution in 2.0 (§4.1).

Note that static and automatic (stack allocated) objects were always possible and that the most effective memory management techniques relied heavily on such objects. The string class was a typical example, here `String` objects are typically on the stack so that they require no explicit memory management and the free store they rely on is managed exclusively and invisibly to the user by the String member functions.

3.3.7 Type Checking

The C++ type checking rules were the result of experiments with the C with Classes. All function calls are checked at compile time. The checking of trailing arguments can be suppressed by explicit specification in a function declaration. This is essential to allow C's `printf()`:

```
int printf(const char* ...); // accept any argument after
                             // the initial character string

// ...

printf("date: %s %d 19%d\n",month,day,year); // maybe right
```

Several mechanisms were provided to alleviate the withdrawal symptoms that many C programmers feel when they first experience strict checking. Overriding type checking using the ellipsis was the most drastic and least recommended of those. Function name overloading (§3.3.3) and default arguments [Stroustrup,1986b] made it possible to give the appearance of a single function taking a variety of argument lists without compromising type safety. The stream I/O system demonstrates that the weak checking wasn't necessary even for I/O (see §5.3.1).

3.4 Relationship to Classic C

With the introduction of a separate name, C++, and the writing of a C++ reference manual [Stroustrup,1984a] compatibility with C became an issue of major importance and a point of controversy.

Also, in late 1983 the branch of Bell Labs that developed and supported UNIX and produced AT&T's 3B series of computers became interested in C++ to the point where they were willing to put resources into the development of C++ tools. Such development was necessary for the evolution of C++ from a one man show to a language that a corporation could base critical projects on. Unfortunately, it also implied that development management needed to consider C++.

The first demand to emerge from development management was that of 100% compatibility with C. The ideal of C compatibility is quite obvious and reasonable, but the reality of programming isn't that simple. For starters, with which C should C++ be compatible? C dialects abounded, and though ANSI C was emerging, it was still years from having a stable definition and its definition allowed many dialects. Naturally, the average user who wanted C compatibility insisted that C++ should be compatible with the local C dialect. This was an important practical problem and a great concern to me and my friends. It seemed far less of a concern to business−oriented managers and salesmen who either didn't quite understand the technical details or would like to use C++ to tie users into their software and/or hardware.

Another side of the compatibility issue was more critical: ''In which ways must C++ differ from C to meet its fundamental goals?'' and also ''In which ways must C++ be compatible with C to meet its fundamental goals?'' Both sides of the issue are important and revisions were made in both directions during the transition from C with Classes to C++ as shipped as release 1.0. Slowly and painfully an agreement emerged that there would be no gratuitous incompatibilities between C++ and ANSI C (when it became a standard) [Stroustrup,1986b] but that there was such a thing as an incompatibility that was not gratuitous. Naturally, the concept of ''gratuitous incompatibilities'' was a topic of much debate and took up a disproportional part of my time and effort. This principle has lately been known as ''C++: As close to C as possible – but no closer.'' after the title of a paper by Andrew Koenig and me [Koenig,1989].

Some conclusions about modularity and how a program is composed out of separately compiled parts were explicitly reflected in the original C++ reference manual [Stroustrup,1984a]:

[a] Names are private unless they are explicitly declared public.

[b] Names are local to their file unless explicitly exported from it.

[c] Static type is checked unless explicitly suppressed.

[d] A class is a scope (implying that classes nest properly).

Point [a] doesn't affect C compatibility but [b], [c], [d] imply incompatibilities:

[1] The name of a non-local C function or object is by default accessible from other compilation units,

[2] C functions need not be declared before use and calls are by default not type checked, and

[3] C structure names don't nest (even when they are lexically nested).

In addition,

[4] C++ has a single name space whereas C had a separate name space for ''structure tags'' (§2.4.5).

The ''compatibility wars'' now seem petty and boring, but some of the underlying issues are still unresolved and we are still struggling with them in the ANSI/ISO committee. I strongly suspect that the reason the compatibility wars were drawn out and curiously inconclusive was that we never quite faced the deeper issues related to the differing goals of C and C++ and saw compatibility as a set of separate issues to be resolved individually.

Typically, the least fundamental issue [4] ''name spaces'' took up the most effort, but was eventually resolved [Ellis,1990].

I had to compromise the notion of a class as a scope [3] and accept the C ''solution'' to be allowed to ship release 1.0. One practical problem was that I had never realized that a C struct didn't constitute a scope so that examples like this

```
struct outer {
        struct inner {
                int i;
        };
        int j;
};

struct inner a = { 1 };
```

are legal C. When the issue came up towards the end of the compatibility wars I didn't have time to fathom the implications of the C ''solution'' and it was much easier to agree than to fight the issue. Later, after many technical problems and much discontent from users, nested class scopes were re-introduced into C++ in 1989 [Ellis,1990].

After much hassle, C++'s stronger type checking of function calls was accepted (unmodified). An implicit violation of the static type system is the original example of a C/C++ incompatibility that is not gratuitous. As it happens, the ANSI C committee adopted a slightly weaker version of C++'s rules and notation on this point and declared uses that don't conform to the C++ rules obsolete.

On issue [1], I had to accept the C rule that global names are by default accessible from other compilation units. There simply wasn't any support for the more restrictive C++ rule. This meant that C++, like C, lacks an effective mechanism for expressing modularity above the level of the class and the file. This has led to a series of complaints and the ANSI/ISO committee is now looking into several proposals for mechanisms to avoid name space pollution†. However, people – such as Doug McIlroy – who argued that C programmers would not accept a language where every object and function meant to be accessible from another compilation unit had to be explicitly declared as such were probably right at the time and saved me from making a serious mistake. I am now convinced that the original C++ solution wasn't elegant enough anyway.

† In 1994, the C++ standard committee accepted a proposal for a namespace mechanism; see [Stroustrup,1994].

3.5 Tools for Language Design

Theory and tools more advanced than a blackboard have not been given much space in the description of the history of C++. I tried to use YACC (an LALR(1) parser generator) for the grammar work, and was defeated by C's syntax (§2.4.5). I looked at denotational semantics, but was again defeated by quirks in C. Ravi Sethi had looked into that problem and found that he couldn't express the C semantics that way [Sethi,1980]. The main problem was the irregularity of C and the number of implementation-dependent and undefined aspects of a C implementation. Much later, the ANSI/ISO C++ committee had a stream of formal definition experts explain their techniques and tools and give their opinion of the extent to which a genuine formal approach to the definition of C++ would help us in the standards effort. My conclusion is that with the current state of the art, and certainly with the state of the art in the early 1980s, a formal definition of a language that is not designed together with a formal definition method is beyond the ability of all but a handful of experts in formal definition.

This confirms my conclusion at the time. However, that left us at the mercy of imprecise and insufficient terminology. Given that, what could I do to compensate? I tried to reason about new features both on my own and with others to check my logic. However, I soon developed a healthy disrespect for arguments (definitely including my own) because I found that it is possible to construct a plausible logical argument for just about any feature. On the other hand, you simply don't get a useful language by accepting every feature that makes life better for someone. There are far too many reasonable features and no language could provide them all and stay coherent. Consequently, wherever possible, I tried to experiment.

My impression was and is that many programming languages and tools represent solutions looking for problems, and I was determined that my work should not fall into that category. Thus, I follow the literature on programming language and the debates about programming languages primarily looking for ideas for solutions to problems my colleagues and I have encountered in real applications. Other programming languages constitute a mountain of ideas and inspiration – but it has to be mined carefully to avoid featurism and inconsistencies. The main sources for ideas for C++ were Simula, Algol68, and later Clu, Ada, and ML. The key to good design is insight into problems, not the provision of the most advanced features.

3.6 The C++ Programming Language (1st edition)

In the fall of 1983 my next door neighbor at work, Al Aho, suggested that I write a book on C++ structured along the lines of Brian Kernighan and Dennis Ritchie's ''The C Programming Language'' based on my published papers, internal memoranda, and the C++ reference manual. Completing the book took nine months.

The preface mentions the people who had by then contributed most to C++: Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Larry Rosler, Jerry Schwarz, and Jon Shopiro. My criteria for adding a person to that list was that I was able to identify a specific C++ feature that the person has caused to be added.

The book's opening line ''C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer'' was deleted twice by reviewers who refused to believe that the purpose of programming language design could be anything but some serious mutterings about productivity, management, and software engineering. However,

> ''C++ was designed primarily so that the author and his friends would not have to program in assembler, C, or various modern high-level languages. Its main purpose is to make writing good programs easier and more pleasant for the individual programmer.''

This was the case whether those reviewers were willing to believe it or not. The focus of my work is the person, the individual (whether part of a group or not), the programmer. This line of reasoning has been strengthened over the years and is even more prominent in the 2nd edition

[Stroustrup,1991].

''The C++ Programming Language'' was the definition of C++ and the introduction to C++ for an unknown number of programmers and its presentation techniques and organization (borrowed with acknowledgements if not always sufficient skill from ''The C Programming Language'') have become the basis for an almost embarrassing number of articles and books. It was written with a fierce determination not to preach any particular programming technique. In the same way as I feared to build limitations into the language out of ignorance and misguided paternalism, I didn't want the book to turn into a ''manifesto'' for my personal preferences.

3.7 The ''whatis?'' Paper

Having shipped release 1.0 and sent the camera ready copy of the book to the printers I finally found time to re-consider larger issues and to document overall design issues. Just then Karel Babcisky (the chairman of the Association of Simula Users) phoned from Oslo with an invitation to give a talk on C++ at the 1986 ASU conference in Stockholm. Naturally, I wanted to go but I was worried that presenting C++ at a Simula conference would be seen as a vulgar example of self-advertisement and an attempt to steal users away from Simula. After all, I said, C++ is not Simula so why would Simula-users want to hear about it. Karel replied ''Ah, we are not hung up on syntax.'' This provided me with an opportunity to write not only about what C++ was but also what it was supposed to be and where it didn't measure up to those ideals. The result was the paper *What is ''Object-Oriented Programming''?* [Stroustrup86] that I presented to the ASU conference in Stockholm.

The significance of this paper is that it is the first exposition of the set of techniques that C++ was aiming to provide support for. All previous presentations, to avoid dishonesty and hype, had been restricted to describe what features were already implemented and in use. The ''whatis paper'' defined the set of problems I thought a language supporting data abstraction and object-oriented programming ought to solve and gave examples of language features needed.

The result was a re-affirmation of the importance of the ''multi-paradigm'' nature of C++:

> ''Object-oriented programming is programming using inheritance. Data abstraction is programming using user-defined types. With few exceptions, object-oriented programming can and ought to be a superset of data abstraction. These techniques need proper support to be effective. Data abstraction primarily needs support in the form of language features and object-oriented programming needs further support from a programming environment. To be general purpose, a language supporting data abstraction or object-oriented programming must enable effective use of traditional hardware.''

The importance of static type checking was also strongly emphasized. In other words, C++ follows the Simula rather than the Smalltalk model of inheritance and type checking:

> ''a Simula or C++ class specifies a fixed interface to a set of objects (of any derived class) whereas a Smalltalk class specifies an initial set of operations for objects (of any subclass). In other words, a Smalltalk class is a minimal specification and the user is free to try operations not specified whereas a C++ class is an exact specification and the user is guaranteed that only operations specified in the class declaration will be accepted by the compiler.''

This has deep implications on the way one design systems and on what language facilities are needed. A dynamically typed language such as Smalltalk simplifies the design and implementation of libraries by postponing type checking to run time. For example (using C++ syntax):

```
stack cs;
cs.push(new Saab900);
cs.pop()->takeoff();  // Oops! Run time error:
                      // a car does not have a takeoff method.
```

This delayed type error detection was considered unacceptable for C++, yet there had to be a way of matching the notational convenience and the standard libraries of a dynamically typed

language. The notion of parameterized types was presented as the (future) solution for that problem in C++:

```
stack(plane*) cs;

cs.push(new Saab37b);  // ok a Saab37b is a plane
cs.push(new Saab900);  // error, type mismatch:
                       // car passed, plane* expected

cs.pop()->takeoff();   // no run-time check needed
cs.pop()->takeoff();   // no run-time check needed
```

The key reason for considering compile time detection of such problems essential was the observation that C++ is often used for programs executing where no programmer is present. Fundamentally, the notion of static type checking was seen as the best way of providing as strong guarantees as possible for a program rather than merely a way of gaining run-time efficiency.

The ''whatis'' paper lists three aspects in which C++ was deficient:

[1] ''Ada, Clu, and ML support parameterized types. C++ does not; the syntax used here is simply devised as an illustration. Where needed, parameterized classes are ''faked'' using macros. Parameterized classes would clearly be extremely useful in C++. They could easily be handled by the compiler, but the current C++ programming environment is not sophisticated enough to support them without significant overhead and/or inconvenience. There need not be any run-time overheads compared with a type specified directly.''

[2] ''As programs grow, and especially when libraries are used extensively, standards for handling errors (or more generally: ''exceptional circumstances'') become important. Ada, Algol68, and Clu each support a standard way of handling exceptions. Unfortunately, C++ does not. Where needed exceptions are ''faked'' using pointers to functions, ''exception objects,'' ''error states'', and the C library `signal` and `longjmp` facilities. This is not satisfactory in general and fails even to provide a standard framework for error handling.''

[3] ''Given this explanation it seems obvious that it might be useful to have a class `B` inherit from two base classes `A1` and `A2`. This is called multiple inheritance''

All three facilities were linked to the need to provide better (that is, more general, more flexible) libraries. All are now available in C++. Note that adding multiple inheritance and templates was considered as early as 1982 [Stroustrup,1982].

4 C++ Release 2.0

Now (Mid 1986) the course for C++ was set for all who cared to see. The key design decisions were made. The direction of the future evolution was set with the aim for parameterized types, multiple inheritance, and exception handling. Much experimentation and adjustment based on experience was needed, but the glory days were over. C++ had never been silly putty, but there was now no real possibility for radical change. For good and bad, what was done was done. What was left was an incredible amount of solid work. At this point C++ had about 2,000 users worldwide.

This was the point where the plan – as originally conceived by Steve Johnson and me – was for a development and support organization to take over the day-to-day work on the tools (primarily Cfront), thus freeing me to work on the new features and the libraries that was expected to depend on them. This was also the point where I expected first AT&T and then others would start to build compilers and other tools to eventually make Cfront redundant.

Actually, they had already started, but the good plan was soon derailed due to management indecisiveness, ineptness, and lack of focus. A project to develop a brand new C++ compiler diverted attention and resources from Cfront maintenance and development. A plan to ship a release 1.3 in early 1988 completely fell through the cracks. The net effect was that we had to

wait until June 1989 for release 2.0, and that even though 2.0 was significantly better than release 1.2 in almost all ways, 2.0 did not provide the language features outlined in the ''whatis paper'' and (consequently) a significantly improved and extended library wasn't part of it.

Many of the people who influenced C with Classes and the original C++ continued to help with the evolution in various ways. Phil Brown, Tom Cargill, Jim Coplien, Steve Dewhurst, Keith Gorlen, Laura Eaves, Bob Kelley, Brian Kernighan, Andy Koenig, Archie Lachner, Stan Lippman, Larry Mayka, Doug McIlroy, Pat Philip, Dave Prosser, Peggy Quinn, Roger Scott, Jerry Schwarz, Jonathan Shopiro, and Kathy Stark were explicitly acknowledged in [Stroustrup,1989b].

Stability of the language definition and its implementation was considered essential. The features of 2.0 were fairly simple modifications of the language based on experience with the 1.* releases. The most important aspect of release 2.0 was that it increased the generality of the individual language features and improved their integration into the language.

4.1 Feature Overview

The main features of 2.0 were first presented in [Stroustrup,1987c] and summarized in the revised version of that paper [Stroustrup,1989b] that accompanied 2.0 as part of its documentation:

[1] multiple inheritance,
[2] type-safe linkage,
[3] better resolution of overloaded functions,
[4] recursive definition of assignment and initialization,
[5] better facilities for user-defined memory management,
[6] abstract classes,
[7] static member functions,
[8] `const` member functions,
[9] `protected` members (first provided in release 1.2),
[10] overloading of operator `->`, and
[11] pointers to members (first provided in release 1.2).

Most of these extensions and refinements represented experience gained with C++ and couldn't have been added earlier without more foresight than I possessed. Naturally, integrating these features involved significant work, but it was most unfortunate that this was allowed to take priority over the completion of the language as outlined in the ''whatis'' paper.

Most features enhanced the safety of the language in some way or other. Cfront 2.0 checked the consistency of function types across separate compilation units (type-safe linkage), made the overload resolution rules order independent, and also ensured that more calls were considered ambiguous. The notion of `const` was made more comprehensive, pointers to members closed a loophole in the type system, and provided explicit class-specific memory allocation and deallocation operations to make the error-prone ''assignment to `this`'' technique redundant.

To some people, the most important ''feature'' of release 2.0 wasn't a feature at all but a simple space optimization. From the beginning, the code generated by Cfront tended to be pretty good. As late as 1992, Cfront generated the fastest running code in a benchmark used to evaluate C++ compilers on a Sparc. There have been no significant improvements in Cfront's code generation since Release 1.0. However, release 1.* was wasteful because each compilation unit generated its own set of virtual function tables for all the classes used in that unit. This could lead to megabytes of waste. At the time (about 1984), I considered the waste necessary in the absence of linker support and asked for such linker support. By 1987 that linker support hadn't materialized. Consequently, I re-thought the problem and solved it by the simple heuristic of laying down the virtual function table of a class right next to its first non-virtual non-inline function.

4.2 Multiple Inheritance

In most people's minds multiple inheritance, the ability to have two or more direct base classes, is *the* feature of 2.0. I disagreed at the time because I felt that the sum of the improvements to the type system were of far greater practical importance. Also, adding multiple inheritance in 2.0 was a mistake. Multiple inheritance belongs in C++ but is far less important than parameterized types. As it happened, parameterized types in the form of templates only appeared in release 3.0. There were a couple of reasons for choosing to work on multiple inheritance at the time: The design was further advanced and the implementation could be done within Cfront. Another factor was purely irrational. Nobody doubted that I could implement templates efficiently. Multiple inheritance, on the other hand, was widely supposed to be very difficult to implement efficiently. Thus multiple inheritance seemed more of a challenge and since I had considered it as early as 1982 and found a simple and efficient implementation technique in 1984 I couldn't resist the challenge. I suspect that this is the only case where fashion affected the sequence of events.

In September 1984, I presented the C++ operator overloading mechanism at the IFIP WG2.4 conference in Canterbury [Stroustrup,1984c]. There, I met Stein Krogdahl from the University of Oslo who was just finishing a proposal for adding multiple inheritance to Simula [Krogdahl,1984]. His ideas became the basis for the implementation of ordinary multiple base classes in C++. He and I later found out that the proposal was almost identical to an idea for providing multiple inheritance in Simula that had been considered by Ole-Johan Dahl in 1966 and rejected because it would have complicated the Simula garbage collector [Dahl,1988].

The original and fundamental reason for considering multiple inheritance was simply to allow two classes to be combined into one in such a way that objects of the resulting class would behave as objects of either base class [Stroustrup,1986c]:

> ''A fairly standard example of the use of multiple inheritance would be to provide two library classes `displayed` and `task` for representing objects under the control of a display manager and co-routines under the control of a scheduler, respectively. A programmer could then create classes such as
>
> ```
> class my_displayed_task : public displayed, public task {
>         // ...
> };
>
> class my_task : public task {  // not displayed
>         // ...
> };
>
> class my_displayed : public displayed  {   // not a task
>         // ...
> };
> ```
>
> Using (only) single inheritance only two of these three choices would be open to the programmer.''

The implementation requires little more than remembering the relative offsets of the `task` and `displayed` objects in a `my_displayed_task` object. All the gory implementation details were explained in [Stroustrup,1987a]. In addition, the language design must specify how ambiguities are handled and what to do if a class is specified as a base class more than once in a derived class:

> ''Ambiguities are handled at compile time:

```
class A { public: void f(); /* ... */ };
class B { public: void f(); /* ... */ };
class C : public A, public B {  /* no f() ... */ };

void g() {
        C* p;
        p->f(); // error: ambiguous
}
```

In this, C++ differs from the object-oriented Lisp dialects that support multiple inheritance.'' Basically, I rejected all forms of dynamic resolution beyond the use of virtual functions as unsuitable for a statically typed language under severe efficiency constraints. Maybe, I should at this point have revived the notion of `call` and `return` functions (§2.4.8) to mimic the CLOS `:before` and `:after` methods. However, people were already worrying about the complexity of the multiple inheritance mechanisms and I am always reluctant to re-open old wounds.

Multiple inheritance in C++ became controversial [Cargill,1991] [Carroll,1991] [Waldo,1991] [Sakkinen,1992] for several reasons. The arguments against it centered around the real and imaginary complexity of the concept, the utility of the concept, and the impact of multiple inheritance on other extensions and tool building. In addition, proponents of multiple inheritance can and do argue over exactly what multiple inheritance is supposed to be and how it is best supported in a language. I think – as I did then – that the fundamental flaw in these arguments is that they take multiple inheritance far too seriously. Multiple inheritance doesn't solve all of your problems, but it doesn't need to because it is quite cheap, and sometimes it is very convenient to have. Grady Booch [Booch,1991] expresses a slightly stronger sentiment: ''Multiple inheritance is like a parachute, you don't need it very often, but when you do it is essential.''

4.3 Abstract Classes

The very last feature added to 2.0 before it shipped was abstract classes. Late modification to releases are never popular and late changes to the definition of what will be shipped are even less so. I remember that several members of management thought I had lost contact with the real world when I insisted on this feature.

A common complaint about C++ was (and is) that private data is visible and that when private data is changed then code using that class must be recompiled. Often, this complaint is expressed as ''abstract types in C++ aren't really abstract.'' What I hadn't realized was that many people thought that because they *could* put the representation of an object in the private section of a class declaration then they actually *had to* put it there. This is clearly wrong (and that is how I failed to spot the problem for years). If you don't want a representation in a class, thus making the class an interface only, then you simply delay the specification of the representation to some derived class and define only virtual functions. For example, one can define a `set` of `T` pointers like this:

```
class set {
public:
    virtual void insert(T*);
    virtual void remove(T*);

    virtual int is_member(T*);

    virtual T* first();
    virtual T* next();

    virtual ~set() { }
};
```

This provides all the information that people need to use a `set` except that whoever actually

creates a `set` must know something about how some particular kind of `set` is represented. For example, given:

```
class slist_set : public set, private slist {
    slink* current_elem;
public:
    void insert(T*);
    void remove(T*);

    int is_member(T*);

    virtual T* first();
    virtual T* next();

    slist_set() : slist(), current_elem(0) { }
};
```

we can create `slist_set` objects that can be used as `sets` by users that have never heard of a `slist_set`.

The only problem was that in C++ as defined before 2.0 there was no explicit way of saying ‘‘The `set` class is just an interface: its functions need not be defined, it is an error to create objects of class `set`, and anyone who derives a class from `set` must define the virtual functions specified in `set`.’’ Release 2.0 allowed a class to be declared explicitly *abstract* by declaring one or more of its virtual functions ‘‘pure’’ using the syntax `=0`:

```
class set {                         // abstract class
public:
    virtual void insert(T*) = 0;  // pure virtual function
    virtual void remove(T*) = 0;

    // ...
};
```

The `=0` syntax wasn’t exactly brilliant, but it expresses the desired notion of a pure virtual function in a way that is terse and fits the use of `0` to mean ‘‘nothing’’ or ‘‘not there’’ in C and C++. The alternative, introducing a new keyword, say `pure`, wasn’t an option. Given the opposition to abstract classes as a ‘‘late and unimportant change,’’ I would never simultaneously have overcome the traditional, strong, widespread, and emotional opposition to new keywords in parts of the C and C++ community.

The importance of the abstract class concept is that it allows a cleaner separation between a user and an implementor than is possible without it. This limits the amount of recompilation necessary after a change and also the amount of information necessary to compile an average piece of code. By decreasing the coupling between a user and an implementor, abstract classes provide an answer to people complaining about long compile times and also serve library providers who must worry about the impact to users of changes to a library implementation. I had unsuccessfully tried to explain these notions in [Stroustrup,1986b]. With an explicit language feature supporting abstract classes I was much more successful [Stroustrup,1991].

5 The Explosion in Interest and Use

C++ was designed to serve users. It was not an academic experiment to design the perfect programming language. Nor was it a commercial product meant to enrich its developers. Thus to fulfill its purpose C++ had to have users – and it had:

C++ use	
Date	estimated number of users
Oct 1979	1
Oct 1980	16
Oct 1981	38
Oct 1982	85
Oct 1983	??+2 (no Cpre count)
Oct 1984	??+50 (no Cpre count)
Oct 1985	500
Oct 1986	2,000
Oct 1987	4,000
Oct 1988	15,000
Oct 1989	50,000
Oct 1990	150,000
Oct 1991	400,000

In other words, the C++ user population doubled every 7.5 months or so. These are conservative figures. The actual number of C++ users has never been easy to count. Firstly, there are implementations such as GNU's G++ and Cfront shipped to universities for which no meaningful records can be kept. Secondly, many companies, both tools suppliers and end-users, treat the number of their users and the kind of work they do like state secrets. However, I always had many friends, colleagues, contacts, and many compiler suppliers who were willing to trust me with figures as long as I used them in a responsible manner. This enabled me to estimate the number of C++ users. These estimates are created by taking the number of users reported to me or estimated based on personal experience, rounding them all down, adding them, and then rounding down again. These number are the estimates made at the time and not adjusted in any way. To support the claim that these figures are conservative, I can mention that Borland, the largest single C++ compiler supplier, publicly stated that it had shipped 500,000 compilers by October 1991.

Early users had to be gained without the benefit of traditional marketing. Various forms of electronic communication played a crucial role in this. In the early years most distribution and all support was done using email and relatively early on newsgroups dedicated to C++ were created (*not* at the initiative of Bell Labs employees) that allowed a wider dissemination of information about the language, techniques, and the current state of tools. These days this is fairly ordinary, but in 1981 it was relatively new. I think that only the spread of Interlisp over the Arpanet provides a contemporary parallel.

Later, more conventional forms of communication and marketing arose. After AT&T released Cfront 1.0 some resellers, notably Glockenspiel in Ireland and their US distributor Oasys (later part of Green Hills) started some minimal advertising in 1986, and when independently developed C++ compilers such as Oregon Software's C++ Compiler (developed by Mike Ball at Tau-Metric Software in San Diego) and Zortech's C++ Compiler (developed by Walter Bright in Seattle) appeared 'C++' became a common sight in ads (from about 1988).

5.1 Conferences

In 1987 USENIX, the UNIX Users' association, took the initiative to hold the first conference specifically devoted to C++. Thirty papers were presented to 214 people in Santa Fe, NM in November of 1987.

The Santa Fe conference set a good example for future conferences with a mix of papers on applications, programming and teaching techniques, ideas for improvements to the language, libraries, and implementation techniques. Notably for a USENIX conference, there were papers

on C++ on the Apple MAC, OS/2, the Connection machine, and for implementing non-UNIX operating systems (for example, CLAM [Call,1987] and Choices [Campbell,1987]). The NIH library [Gorlen,1987] and the Inverviews library [Linton,1987] also made their public debut in Santa Fe. An early version of what became Cfront 2.0 was demonstrated and I gave the first public presentation of its features. The USENIX C++ conferences continue as the primary technically and academically oriented C++ conference. The proceedings from these conferences are among the best reading about C++ and its use.

In addition to the USENIX C++ conferences, there are now many commercial and semi-commercial conferences devoted to C++, to C including C++, and to Object-Oriented Programming.

5.2 Journals and Books

By 1991 there were more than 60 books on C++ available in English alone and both translations and locally written books available in languages such as Chinese, Danish, French, German, and Japanese. Naturally, the quality varies enormously.

The first journal devoted to C++, ''*The C++ Report*'' from SIGS publications, started publishing in January 1989 with Rob Murray as its editor, A larger and glossier quarterly ''*The C++ Journal*'' appeared in the spring of 1991. In addition there are several newsletters controlled by C++ tools suppliers and many journals such as *Computer Language, The Journal of Object-Oriented Programming, Dr. Dobbs Journal, The C Users' Journal*, run regular columns or features on C++. Andrew Koenig's column in JOOP is particularly consistent in its quality and lack of hype.

Newsgroup and bulletin boards such as comp.lang.c++ on usenet and c.plus.plus on BIX also produced tens of thousands of messages over the years to the delight and despair of the readers. Keeping up with what is written about C++ is currently more than a full time job.

5.3 Libraries

The very first real code to be written in C with Classes was the task library [Stroustrup,1980b] providing Simula-like concurrency for simulation. The first real programs were simulations of network traffic, circuit board layout, etc. using the task library. The task library is still heavily used today. The standard C library was available from C++ – without overhead or complication compared with C – from day one. So are all other C libraries. Classical data types such as character strings, range checked arrays, dynamic arrays, and lists were among the examples used to design C++ and test its early implementations.

The early work with container classes such as list and array were severely hampered by the lack of support for a way of expressing parameterized types in C with Classes and in C++ up until version 3.0. In the absence of proper language support, (later provided in the form of templates (§6.3) we had to made do with macros. The best that can be said for the C preprocessor's macro facilities is that it allowed us to gain experience with parameterized types and support individual and small group use.

Much of the work on designing classes was done in cooperation with Jonathan Shopiro who in 1983 produced list and string classes that saw wide use within AT&T and are the basis for the classes currently found in the ''Standard Components'' library that was developed in Bell labs and is now sold by USL. The design of these early libraries interacted directly with the design of the language and in particular with the design of the overloading mechanisms.

5.3.1 The Stream I/O Library

C's `printf` family of functions is an effective and often convenient I/O mechanism. It is not, however, type safe or extensible to user-defined types (classes). Consequently, I started looking for a type safe, terse, extensible, and efficient alternative to the `printf` family. Part of the inspiration came from the last page and a half of the Ada Rationale [Ichbiah,1979], which is an argument that you cannot have a terse and type-safe I/O library without special language features to support it. I took that as a challenge. The result was the stream I/O library that was first implemented in 1984 and presented in [Stroustrup,1985]. Soon after, Dave Presotto reimplemented the stream library without changing the interfaces.

To introduce stream I/O this example was considered:

```
fprintf(stderr,"x = %s\n",x);
```

Because `fprintf()` relies on unchecked arguments that are handled according to the format string at run time this is not type safe and

> ''had `x` been a user-defined type like `complex` there would have been no way of specifying the output format of `x` in the convenient way used for types ''known to `printf()`'' (for example, `%s` and `%d`). The programmer would typically have defined a separate function for printing complex numbers and then written something like this:
>
> ```
> fprintf(stderr,"x = ");
> put_complex(stderr,x);
> fprintf(stderr,"\n");
> ```
>
> This is inelegant. It would have been be a major annoyance in C++ programs that use many user-defined types to represent entities that are interesting/critical to an application.
>
> Type-security and uniform treatment can be achieved by using a single overloaded function name for a set of output functions. For example:
>
> ```
> put(stderr,"x = ");
> put(stderr,x);
> put(stderr,"\n");
> ```
>
> The type of the argument determines which ''put function'' will be invoked for each argument. However, this is too verbose. The C++ solution, using an output stream for which `<<` has been defined as a ''put to'' operator, looks like this:
>
> ```
> cerr << "x = " << x << "\n";
> ```
>
> where `cerr` is the standard error output stream (equivalent to the C `stderr`). So, if `x` is an `int` with the value `123`, this statement would print
>
> ```
> x = 123
> ```
>
> followed by a newline onto the standard error output stream.
>
> This style can be used as long as `x` is of a type for which operator `<<` is defined, and a user can trivially define operator `<<` for a new type. So, if `x` is of the user-defined type `complex` with the value `(1,2.4)`, the statement above will print
>
> ```
> x = (1,2.4)
> ```
>
> on `cerr`.
>
> The stream I/O facility is implemented exclusively using language features available to every C++ programmer. Like C, C++ does not have any I/O facilities built into the language. The stream I/O facility is provided in a library and contains no ''extra-linguistic magic''.''

The idea of providing an output operator rather than a named output function was suggested by Doug McIlroy. This requires operators that return their left-hand operand for use by further operations.

In connection with release 2.0, Jerry Schwarz reimplemented and partially redesigned the streams library to serve a larger class of applications and to be more efficient for file I/O. A significant improvement was the use of Andrew Koenig's idea of manipulators [Stroustrup,1991] to control formatting details such as the precision used for floating point output. Experience with streams was a major reason for the change to the basic type system and to the overloading rules to allow `char` values to be treated as characters rather than small integers the way they are in C. For example:

```
char ch = 'b';
cout << 'a' << ch;
```

would in Release 1.* output a string of digits reflecting the integer values of the characters `a` and `b` whereas Release 2.* outputs `ab` as one would expect.

5.3.2 Other Libraries

There were and are many other significant C++ libraries. These will be mentioned only briefly here because even though they were essential to their users they did not affect the development of C++ significantly. They are, however, most significant to their users and most users' view of C++ is strongly affected or even dominated by a library.

The most significant early libraries was Keith Gorlen's [Gorlen,1990] NIH class library that provides a Smalltalk-like set of classes and Mark Linton's Interviews library [Linton,1987] that makes use of the X windows system convenient from C++. GNU C++ (G++) comes with a library designed by Doug Lea that is distinguished by heavy use of abstract base classes. Rogue Wave and Dyad supply large sets of libraries primarily aimed at scientific uses. Glockenspiel has for years supplied libraries for various commercial uses. Rational ships a C++ version of ''The Booch Components'' that was originally designed for and implemented in Ada by Grady Booch. Grady Booch and Mike Vilot designed and implemented the C++ version. The Ada version is 150,000 non-commented source lines compared to the C++ version's 10,000 lines – inheritance combined with templates can be a very powerful mechanism for organizing libraries without loss of performance or clarity.

This is only a very short list of early libraries to indicate the diversity of C++ libraries. Many more libraries exists. In particular, most tools suppliers provide foundation libraries for their users. It seems that the ''software components'' industry that pundits have promised for years – and bemoaned the lack of – has finally come into existence.

5.4 Compilers

The Santa Fe conference (§5.1) marked the announcement of the second wave of C++ implementations. Steve Dewhurst described the architecture of a compiler he and others were building in AT&T's Summit facility, Mike Ball presented some ideas for what became the TauMetric C++ compiler (more often known as the Oregon Software C++ compiler), and Mike Tiemann gave a most animated and interesting presentation of how the GNU G++ he was building would do just about everything and put all other C++ compiler writers out of business. The new AT&T C++ compiler never materialized; GNU C++ version 1.13 was first released in December 1987; and TauMetric C++ first shipped in January 1988.

Until June 1988 all C++ compiler on PCs were Cfront ports. Then Zortech started shipping their compiler. The appearance of Walter Bright's compiler made C++ ''real'' for many PC-oriented people for the first time. More conservative people reserved their judgement until the Borland C++ compiler in May 1990 or even Microsoft's C++ compiler in March 1992. DEC released their first independently developed C++ compiler in February 1992 and IBM released their first independently developed C++ compiler in May 1992. In all there are now more than a dozen independently developed C++ compilers.

In addition to these compilers, Cfront ports seemed to be everywhere. In particular, Sun, HP,

Centerline, ParcPlace, Glockenspiel, and Comeau Computing ship Cfront-based products on just about any platform.

5.5 Tools and Environments

C++ was designed to be a viable language in a tool-poor environment. This was partly a necessity because of the almost complete lack of resources in the early years and the relative poverty later on. It was also a conscious decision to allow simple implementations and in particular simple porting of implementations.

C++ programming environments are now emerging that are a match for the environments habitually supplied with other object-oriented languages. For example, ObjectWorks for C++ from ParcPlace is essentially the best Smalltalk program development environment adapted for C++, and Centerline C++ (formerly Saber C++) is an interpreter-based C++ environment inspired by the interlisp environment. This gives C++ programmers the option of using the more whizzy, more expensive, and often more productive environments that have previously only been available for other languages and/or as research toys. An environment is a framework in which tools can cooperate. There is now a host of such environments for C++: Most C++ implementations on PCs are compilers embedded in a framework of editors, tools, file systems, standard libraries, etc. MacApp and the Mac MPW is the Apple Mac version of that, ET++ is a public domain version in the style of the MacApp. Lucid's Energize and HP's Softbench are yet other examples.

5.6 Commercial Competition

Commercial competitors were largely ignored and the C++ language was developed according to the original plan, its own internal logic, and the experience of its users. There was (and is) always much discussion among programmers, in the press, at conferences, and on the electronic bulletin boards about which language ''is best'' and which language will ''win'' in some sort of competition for users. Personally, I consider much of that debate misguided and uninformed, but that doesn't make the issues less real to a programmer, manager, or professor who has to choose a programming language for his or her next project. For good and bad, people debate programming languages with an almost religious fevor and often consider the choice of programming language the most important choice of a project or organization.

In the early years, Modula-2 [Wirth,1982] was by many considered a competitor to C++. However, until the commercial release of C++ in 1985, C++ could hardly be considered a competitor to any language, and by then Modula-2 seemed to me to have been largely outcompeted by C. Later it was popular to speculate about whether C++ or Objective C [Cox,1986] was to be ''*the* Object-Oriented C.'' Ada [Ichbiah,1979] was often a possible choice of organizations who might use C++. In addition, Smalltalk [Goldberg,1983] and some object-oriented variant of Lisp [Kiczales,1992] would often be considered for applications that did not require hard-core systems work or maximum performance. Lately, some people have been comparing C++ with Eiffel [Meyer,1988] and Modula-3 [Nelson,1991] for some uses.

My personal view is different. The main competitor to C++ was C. The reason that C++ is the most widely used object-oriented language today is that it was/is the only one that could consistently match C on C's own turf and that allows a transition path from C to a style of system design and implementation based on a more direct mapping between application level concepts and language concepts (usually called ''data abstraction'' or ''object-oriented programming.''). Secondarily many organizations that consider a new programming language have a tradition for the use of an in-house language (usually a Pascal variant) or Fortran. Except for serious scientific computation these languages can be considered roughly equivalent to C when compared with C++.

In the secondary competition between C++ and other newer languages supporting abstraction mechanisms (object-oriented programming languages, languages supporting data abstraction)

C++ was during the early years (1984 to 1989) consistently the underdog as far as marketing was concerned. In particular, AT&T's marketing budget during that period was usually empty and AT&T's total spending on C++ advertising was about $3,000. To this day, most of AT&T's visibility in the C++ arena relies on Bell Labs' traditional policy of encouraging developers and researchers to give talks, write papers, and attend conferences rather than on any deliberate policy to promote C++. Within AT&T, C++ was also a grass-roots movement without money or management clout. Naturally, coming from AT&T Bell Labs helps C++, but that help is earned the hard way by surviving in a large-company environment.

In competition, C++'s fundamental strength is its ability to operate in a traditional environment (social and computer-wise), its run-time and space efficiency, the flexibility of its class concept, its low price, and its non-proprietary nature. Its weaknesses compared to newer languages are some of the uglier parts inherited from C, its lack of spectacular new features (such as built-in data base support), its lack of spectacular program development environments (only lately have C++ environments of the sort people have taken for granted for Smalltalk and Lisp become available for C++), its lack of standard libraries (only lately have major libraries become widely available for C++ – and they are not ''standard''†), and its lack of salesmen to balance the efforts of richer competitors. With C++'s recent dominance in the market the last factor has disappeared. Some C++ salesmen will undoubtedly embarrass the C++ community by emulating some of the sleazy tricks and unscrupulous practices that salesmen and admen have used to attempt to derail C++'s progress.

An important factor, both for and against C++, was the willingness of the C++ community to acknowledge C++'s many imperfections. This openness is reassuring to many that have become cynics from years of experience with the people and products of the software tools industry, but also infuriating to perfectionists and a fertile source for fair and not-so-fair criticism of C++. On balance, I think that tradition of throwing rocks at C++ within the C++ community has been a major advantage. It kept us honest, kept us busy improving the language and its tools, and kept the expectations of C++ users and would-be users realistic.

In competition with traditional languages, C++'s inheritance mechanism was a major plus. In competition with languages with inheritance, C++'s static type checking was a major plus. Of the languages mentioned, only Eiffel and Modula-3 combines the two in a way similar to C++. It is widely assumed that Ada will be revised to include inheritance [Tucker,1992].

C++ was designed to be a systems programming language and a language for applications that had a large ''systems-like'' component. This was the area that my friends and I knew well. The decision not to compromise C++'s strengths in this area to broaden its appeal has been crucial in its success. Only time will tell if this has also compromised its ability to appeal to an even larger audience. I would not consider that a tragedy because I am not among those that think that a single language should be all things to all people and C++ already serves the community it was designed for well. However, I suspect that through the design of libraries C++'s appeal will be very wide.

---

† This problem has now been remedied. The standard C++ library provides standard containers, such as list, vector, map, set, etc. and a library of sorting, searching, etc. algorithms operating on these containers. These containers and algorithms are all templates. This part of the standard library is based on the work of Alex Stepanov [Stepanov,1994]. In addition, the standard library provides a vector type with associated operations to support numeric calculations based on the work of Ken Budge.

6 Standardization

Sometime in 1988 it became clear that C++ would eventually have to be standardized [Stroustrup,1989]. There were now a handful of independent implementations in use or being produced and clearly an effort had to be made to write a more precise and comprehensive definition of the language and also to gain wide acceptance for that definition. At first, formal standardization wasn't considered an option. Many people involved with C++ considered – and still consider – standardization before genuine experience has been gained abhorrent. However, making an improved reference manual wasn't something that could be done by one person (me) in private. Input and feedback from the C++ community was needed. Thus I came upon the idea of re−writing the C++ reference manual and circulating its draft among important and insightful members of the C++ community worldwide.

6.1 The Annotated Reference Manual

At about the same time, the part of AT&T that sold C++ commercially wanted a new and improved C++ reference manual and gave Margaret Ellis the task of writing it. It seemed only reasonable to combine the efforts and produce a single, externally reviewed reference manual. It also seemed obvious to me that publishing this manual with some additional information would help the acceptance of the new definition and make C++ more widely understood. Thus, the Annotated C++ Reference Manual was written ''to provide a firm basis for the further evolution of C++ ... (and) to serve as a starting point for the formal standardization of C++ [Ellis,1990].''

> ''The C++ reference manual alone provides a complete definition of C++, but the terse reference manual style leaves many reasonable questions unanswered. Discussions of what is *not* in the language, *why* certain features are defined as they are, and *how* one might implement some particular feature have no place in a reference manual but are nevertheless of interest to most users. Such discussions are presented as annotations and in the commentary sections.
>
> The commentary also helps the reader appreciate the relationships among different parts of the language and emphasizes points and implications that might have been overlooked in the reference manual itself. Examples and comparisons with C also make this book more approachable than the bare reference manual [Ellis,1990].''

After some minor squabbling with the product people it was agreed that we'd write the ARM (as ''The Annotated C++ Reference Manual'' came to be popularly called) describing the whole of C++, that is with templates and exception handling, rather than as a manual for the subset implemented by the most recent AT&T release. This was important because it clearly established the language itself as different from any one implementation of it. This principle had been present from the very beginning, but needs to be restated often because users and implementors seem to have difficulties remembering it.

Of the ARM, I wrote every word of the reference manual proper except the section on the preprocessor that Margaret Ellis adopted from the C Standard. The annotations were jointly written and partly based on my earlier papers [Stroustrup,1987a] [Stroustrup,1987c] [Stroustrup,1988a] [Stroustrup,1988b].

The reference manual proper of the ARM was reviewed by about a hundred people from two dozen organizations. Most are named in the acknowledgement section of the ARM. In addition, many contributed to the whole of the ARM. The contributions of Brian Kernighan, Andrew Koenig, and Doug McIlroy were specifically noted. The reference manual proper from the ARM was accepted as the basis for the ANSI standardization of C++ in March 1990.

The ARM doesn't attempt to explain the techniques that the language features support. That job was left for the second edition of *The C++ Programming Language* [Stroustrup,1991].

6.2 Minor Features

The ARM presented a few minor features that were not implemented until 2.1 releases from AT&T and other C++ compiler vendors. The most obvious of these were nested classes. I was strongly encouraged to revert to the original definition of nested class scopes by comments from external reviewers of the reference manual. I also despaired over ever getting the scope rules of C++ coherent while the C rule was in place (§3.4).

The ARM allowed people to overload prefix and postfix increment (++) independently. The main impetus for that came from people who wanted ''smart pointers'' that behaved exactly like ordinary pointers except for some added work done ''behind the scenes.''

6.3 Templates

In the original design of C++, parameterized types (templates) were considered but postponed because there wasn't time to do a thorough job of exploring the design and implementation issues. I first presented templates at the 1988 USENIX C++ conference in Denver [Stroustrup,1988b]:

> ''For many people, the largest single problem using C++ is the lack of an extensive standard library. A major problem in producing such a library is that C++ does not provide a sufficiently general facility for defining ''container classes'' such as lists, vectors, and associative arrays.''

There are two approaches for providing such classes/types: One can either rely on dynamic typing and inheritance like Smalltalk does, or one can rely on static typing and a facility for arguments of type *type*. The former is very flexible, but carries a high run-time cost, and more importantly defies attempts to use static type checking to catch interface errors. Therefore, the latter approach was chosen.

A C++ parameterized type is called a class template. A class template specifies how individual classes can be constructed much like the way a class specifies how individual objects can be constructed. A vector class template might be declared like this:

```
template<class T> class vector {
        T* v;
        int sz;
public:
        vector(int);
        T& operator[](int);
        T& elem(int i) { return v[i]; }
        // ...
};
```

The `template <class T>` prefix specifies that a template is being declared and that an argument `T` of type *type* will be used in the declaration. After its introduction, `T` is used exactly like other type names within the scope of the template declaration. Vectors can then be used like this:

```
vector<int> v1(20);
vector<complex> v2(30);

typedef vector<complex> cvec;   // make cvec a synonym for
                                // vector<complex>

cvec v3(40);                    // v2 and v3 are of the same type

v1[3] = 7;
v2[3] = v3.elem(4) = complex(7,8);
```

C++ does not require the user to explicitly ''instantiate'' a template; that is, the user need not specify which versions of a template needs to be generated for particular sets of template

arguments. The reason is that only when the program is complete can it be known what templates need to be instantiated. Many templates will be defined in libraries and many instantiations will be directly and indirectly caused by users that don't even know of the existence of those templates. It therefore seemed unreasonable to require the user to request instantiations (say, by using something like Ada's 'new' operator).

Avoiding unnecessary space overheads caused by too many instantiations of template functions was considered a first order – that is, language level – problem rather than an implementation detail. I considered it unlikely that early (or even late) implementations would be able to look at instantiations of a class for different template arguments and deduce that all or part of the instantiated code could be shared. The solution to this problem was to use the derived class mechanism to ensure code sharing among derived template instances.

The template mechanism is completely a compile and link time mechanism. No part of the template mechanism needs run-time support. This leaves the problem of how to get the classes and functions generated (instantiated) from templates to depend on information known only at run time. The answer was, as ever in C++, to use virtual functions and abstract classes. Abstract classes used in connection with templates also have the effect of providing better information hiding and better separation of programs into independently compiled units.

6.4 Exception Handling

Exceptions were considered in the original design of C++, but were postponed because there wasn't time to do a thorough job of exploring the design and implementation issues. Exceptions were considered essential for error handling in programs composed out of separately designed libraries.

The actual design of the C++ exception mechanism stretched from 1984 to 1989. Andrew Koenig was closely involved in the later iterations and is the co-author (with me) on the published papers [Koenig,1989a] [Koenig,1990] [Koenig,1990]. I also had meetings at Apple, DEC, Microsoft, IBM, Sun, and other places where I presented draft versions of the design and received valuable input. In particular, I searched out people with actual experience with systems providing exception handling to compensate for my personal inexperience in that area. Throughout the design effort there was an increasing influence of systems designers of all sorts and a decrease of input from the language design community. In retrospect, the greatest influence on the C++ exception handling design was the work on fault-tolerant systems started at the University of Newcastle in England by Brian Randell and his colleagues and continued in many places since.

The following assumptions were made for the design:

- Exceptions are used primarily for error handling.
- Exception handlers are rare compared to function definitions.
- Exceptions occur infrequently compared to function calls.

These assumptions, together with the requirement that C++ with exceptions should cooperate smoothly with languages without exceptions, such as C and Fortran, led to a design with multi-level propagation. The view is that not every function should be a fire-wall and that the best error-handling strategies are those where only designated major interfaces are concerned with non-local error handling issues. By allowing multi-level propagation of exceptions C++ loses one aspect of static checking. One cannot simply by looking at a function determine which exceptions it may throw. C++ compensates by providing a mechanism for specifying a list of exceptions that a function may throw.

I concluded that the ability to define groups of exceptions is essential. For example, a user must be able to catch ''any I/O library exception'' without knowing exactly which exceptions those are. Many people, including Ted Goldstein and Peter Deutsch, noticed that most such groups were equivalent to class hierarchies. We therefore adopted a scheme inspired by ML where you throw an object and catch it by a handler declared to accepts objects of that type. This

scheme naturally provides for type-safe transmission of arbitrary amounts of information from a throw point to a handler. For example:

```
class Matherr { /* ... */ };
class Overflow : public Matherr { /* ... */ };
class Underflow : public Matherr { /* ... */ };
class Zerodivide : public Matherr { /* ... */ };
class Int_add_overflow : public Overflow { /* ... */ };
// ...

try {
        f();
}
catch (Overflow& over) {
        // handle Overflow or anything derived from Overflow
}
catch (Matherr& math) {
        // handle any Matherr
}
```

Thus `f()` might be written like this

```
void f() throw(Matherr&) // f() can throw Matherr& exceptions
                         // (and only Matherr& exceptions)
{
        // ...
        if (d == 0) throw Zerodivide();
        // ...
        if ( check(x,y) ) throw Int_add_overflow(x,y);
        // ...
}
```

The `Zerodivide` will be caught by the `Matherr&` handler above and `Int_add_overflow` will be caught by the `Overflow&` handler that might access the operand values `x` and `y` passed by `f()` in the object thrown.

The central point in the exception handling design was the management of resources. In particular, if a function grabs a resource how can the language help the user to ensure that the resource is correctly released upon exit even if an exception occurs? The problem with most solutions are that they are verbose, tedious, potentially expensive and therefore error-prone. However, I noticed that many resources are released in the reverse order of their acquisition. This strongly resembles the behavior of local objects created by constructors and destroyed by destructors. Thus we can handle such resource acquisition and release problems by a suitable use of objects of classes with constructors and destructors. This technique extends to partially constructed objects and thus addresses the otherwise difficult issue of what to do when an error is encountered in a constructor.

During the design the most contentious issue turned out to be whether the exception handling mechanism should support termination semantics or resumption semantics; that is, whether it should be possible for an exception handler to require execution to resume from the point where the exception was thrown. The main resumption vs termination debate took place in the ANSI C++ committee. After a discussion that lasted for about a year, the exception handling proposal as presented in the ARM (that is, with termination semantics) was voted into C++ by an overwhelming majority. The key to that consensus was presentations of experience data based on decades of use of systems that supported both resumption and termination semantics by representatives of DEC, Sun, Texas Instruments, IBM, and others. Basically, every use of resumption had represented a failure to keep separate levels of abstraction disjoint.

The C++ exception handling mechanism is explicitly *not* for handling asynchronous events directly. This view precludes the direct use of exceptions to represent something like hitting a DEL key and the replacement of UNIX signals with exceptions. In such cases, a low-level interrupt routine must somehow do its minimal job and possibly map into something that could trigger an exception at a well-defined point in a programs execution.

As ever, efficiency was a major concern. The C++ exception handling mechanism can be implemented without any run-time overhead to a program that doesn't throw an exception [Stroustrup,1988b]. It is also possible to limit space overhead, but it is hard simultaneously to avoid run-time overhead and code size increases. The first implementations of exception handling as defined in the ARM are just appearing (Spring 1992).

6.5 ANSI and ISO

The initiative to formal (ANSI) standardization of C++ was taken by HP in conjunction with AT&T, DEC, and IBM. Larry Rosler from HP was important in this initiative. The proposal for ANSI standardization was written by Dmitry Lenkov [Lenkov,1989]. Dmitry's proposal cites several reasons for immediate standardization of C++:

- C++ is going through a much faster public acceptance than most other languages.
- Delay ... will lead to dialects.
- Requires a careful and detailed definition providing full semantics ... for each language feature.
- C++ lacks some important features ... exception handling, aspects of multiple inheritance, features supporting parametric polymorphism, and standard libraries.

The proposal also stressed the need for compatibility with ANSI C. The organizational meeting of the ANSI C++ committee, X3J16 took place in December of 1989 in Washington, D.C. and was attended by about 40 people including people who took part in the C standardization, people who by now were ''old time C++ programmers,'' and others. Dmitry Lenkov became its chairman and Jonathan Shopiro became its editor.

The committee now has more than 250 members out of which something like 70 turn up at meetings. The aim of the committee was and is a draft standard for public review in late 1993 or early 1994 with the hope of an official standard about two years later†. This is an ambitious schedule for the standardization of a general-purpose programming language. To compare, the standardization of C took 7 years.

Naturally, standardization of C++ isn't just an American concern. From the start, representatives from other countries attended the ANSI C++ meetings; and in Lund, Sweden, in June 1991 the ISO C++ committee WG21 was convened and the two C++ standards committees decided to hold joint meetings – starting immediately in Lund. Representatives from Canada, Denmark, France, Japan, Sweden, the UK, and USA were present. Notably, the vast majority of these national representatives were actually long-time C++ programmers. The C++ committee had a difficult charter:

[1] The definition of the language must be precise and comprehensive.
[2] C/C++ compatibility had to be addressed.
[3] Extensions beyond current C++ practice had to be considered.
[4] Libraries had to be considered.

On top of that, the C++ community was *very* diverse and totally unorganized so that the standards committee naturally became an important focal point of that community. In the short run, that is actually the most important role for the committee.

C compatibility was the first major controversial issue we had to face. After some – occasionally heated – debate it was decided that 100% C/C++ compatibility wasn't an option. Neither

† We still expect to see a C++ ISO standard in late-1995 to mid-1996.

was significantly decreasing C compatibility. C++ was a separate language and not a strict superset of ANSI C and couldn't be changed to be such a superset without breaking the C++ type system and without breaking millions of lines of C++ code. This decision, often referred to as ''As close to C, but no closer'' after a paper written by Andrew Koenig and me [Koenig1989a], is the same that has been reached over and over again by individuals and groups considering C++ and the direction of its evolution (§3.4).

6.6 Rampant Featurism?

A critical issue was – and is – how to handle the constant stream of proposals for language changes and extensions. The focus of that effort is the extensions working group of which I'm the chairman. It is much easier to accept a proposal than to reject it. You win friends this way and people praise the language for having so many ''neat features.'' Unfortunately, a language made as a shopping list of features without coherence will die so there is no way we could accept even most of the features that would be of genuine help to some section of the C++ community.

So how is the committee doing? We won't really know until the standard appears because there is no way of knowing which, if any, of the backlog of proposals will be accepted. There is some hope of restraint and that accepted features will be properly integrated into the language. Only three new features have been accepted so far: the ''mandated'' extensions (exception handling and templates), and a proposal for relaxing the requirements for return types for overriding functions†.

7 Retrospective

It is often claimed that hindsight is an exact science. It is not. The claim is based on the false assumptions that we know all relevant facts about what happened in the past, that we know the current state of affairs, and that we have a suitably detached point of view from which to judge the past. Typically none of these conditions hold. This makes a retrospective on something as large, complex, and dynamic as a programming language in large scale use hazardous. Anyway, let me try to stand back and answer some hard questions:

[1] Did C++ succeed at what it was designed for?

[2] Is C++ a coherent language?

[3] What was the biggest mistake?

Naturally, the replies to these questions are related. The basic answers are, 'yes,' 'yes,' and 'not shipping a larger library with release 1.0.'

7.1 Did C++ succeed at what it was designed for?

''C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer [Stroustrup1986b].'' In this, it clearly succeeded, especially in the more specific aim of letting reasonably educated and experienced programmers write programs at a higher level of abstraction (''as in Simula'') without loss of efficiency compared to C for applications that were demanding in time, space, inherent complexity, and constraints from the execution environment.

More generally, C++ made object-oriented programming and data abstraction available to the community of software developers that until then had considered such techniques and the languages that supported them such as Smalltalk, Clu, Simula, Ada, OO Lisp dialects, etc., with disdain and even scorn: ''expensive toys unfit for real problems.'' C++ did three things to overcome

† As I write this, the chance of any significant additions to the C++ language by the standard committee is about zero. Many minor extensions was added, but I think that C++ is now a significantly more powerful, pleasent, and coherent language that is was when the standards process started. The major ''new'' extensions was run-time type information, namespaces, and the sum of many minor improvements to templates; see [Stroustrup,1994].

this formidable barrier:

[1] It produced code with run-time and space characteristics that competed head-on with the perceived leader in that field: C. Anything that matches or beats C *must* be fast enough. Anything that doesn't can and will – out of need or mere prejudice – be ignored.

[2] It allowed such code to be integrated into conventional systems and produced on traditional systems. A conventional degree of portability, the ability to coexist with existing code, and the ability to coexist with traditional tools, such as debuggers and editors, was essential.

[3] It allowed a gradual transition to these new programming techniques. It takes time to learn new techniques. Companies simply cannot afford to have significant numbers of programmers unproductive while they are learning. Nor can they afford the cost of failed projects caused by programmers poorly trained and inexperienced in the new techniques failing by overenthusiastically misapplying ideas.

In other words, C++ made object-oriented programming and data abstraction cheap and accessible.

In succeeding, C++ didn't just help ''itself'' and the C++ programmers. It also provided a major impetus to languages that provided different aspects of object-oriented programming and data abstraction. C++ isn't everything to all people and doesn't deliver on every promise ever made about some language or other. It does deliver on its own promises often enough to break down the wall of disbelief that stood in the way of all languages that allowed programmers to work at a higher level of abstraction.

7.2 Is C++ a Coherent Language?

C++ was successful in its own terms and is an effective vehicle for systems development, but is it a good language? Does C++ have an ecological niche now that the barriers of ignorance and prejudice against abstraction techniques have been broken?

Basically, I am happy with the language and quite a few users agree. There are many details I'd like to improve if I could, but the fundamental concept of a statically typed language using classes with virtual functions as the inheritance mechanism and facilities for low-level programming is sound.

7.2.1 What Should and Could Have Been Different?

Given a clean slate, 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, clean separation between language and environment, no direct support for concurrency, ability to match the layout of objects and call sequences for languages such as C and Fortran, C compatibility.

Firstly, I considered and still consider static type checking essential both as a support for good design and secondarily for delivering acceptable run-time efficiency. Were I to design a new language for what C++ is used for 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 [Stroustrup,1990].'' Having both static type checking and dynamic type identification (for example, in the form of virtual function calls) implies some difficult tradeoffs compared to language with only static or only dynamic type checking. The static type model and the dynamic type model cannot be identical and thus there will be some complexity and inelegance that can be avoided by supporting only one type model. However, I wouldn't want to write programs with only one model.

I also still consider a separation between the environment and the language essential. I do not want to use only one language, only one set of tools, and only one operating system. To offer a choice, separation is necessary. However, once the separation exists one can provide different

environments to suit different tastes and different requirements for supportiveness, resource consumption, and portability.

We never have a clean slate. Whatever new we do we must also make it possible for people to make a transition from old tools and ideas to new. Thus, if C hadn't been there for C++ to be almost compatible with then I would have chosen to be almost compatible with some other language.

Should a new language support garbage collection directly, say, like Modula-3 does? If so, could C++ have met its goals had it provided garbage collection? Garbage collection is great when you can afford it. Therefore, the option of having garbage collection is clearly desirable. However, garbage collection can be costly in terms of run time, real-time response, and porting effort (exactly how costly is the topic of much confused debate). Therefore, being forced to pay for garbage collection at all times isn't necessarily a blessing. C++ allows *optional* garbage collection [Ellis,1990], and I expect to see many experiments with garbage collecting C++ implementations in the near future. However, I am convinced (after reviewing the issue many times over the years) that had C++ depended on garbage collection it would have been stillborn.

Should a language have reference semantics for variables (that is a name is really a pointer to an object allocated elsewhere) like Smalltalk or Clu or true local variables like C and Pascal? This question relates to several issues such as co-existence with other languages, compatibility, and garbage collection. Simula dodged the question by having references to class objects (only) and true variables for objects of built-in types (only). Again, I consider it an open issue whether a language could be designed that provided the benefits of both references and true local variables without ugliness. Given a choice between elegance and the benefits of having both references and true local variables, I'll take the two kinds of variables.

7.2.2 What Should Have Been Left Out?

Even [Stroustrup,1980a] voiced concern that C with Classes might have become too large. I think ''a smaller language'' is number one on any wish list for C++; yet people deluge me and the standards committee with extension proposals. The fundamental reason for the size of C++ is that it supports more than one way of writing programs, more than one programming paradigm. From one point of view C++ is really three languages in one: A C-like language plus an Ada-like language, plus a Simula-like language, plus what it takes to integrate those features into a coherent whole.

Brian Kernighan observes that in C there is usually about one way of solving a given problem whereas, in C++ there are more. I conjecture that there typically are more than one way in C but that people don't see them. In C++, there are typically at least three alternatives and experienced people have quite a hard time not seeing them. There always is a design choice but in most languages the language designer has made the choice for you. For C++ I did not; the choice is yours. This is naturally abhorrent to people who believe that there is exactly one right way of doing things. It can also scare beginners and teachers that feel that a good language is one that you can completely understand in a week. C++ is not such a language. It was designed to provide a tool set for a professional and complaining that there are too many features is like the ''layman'' looking into an upholsterer's tool chest and exclaiming that there couldn't possibly be a need for all of those little hammers.

7.2.3 What Should Have Been Added?

As ever, the principle is to add as little as possible. A letter published on behalf of the extensions working group of the C++ standards committee puts it this way [Stroustrup,1992b]:

> ''First, let us try to dissuade you from proposing an extension to the C++ language. C++ is already too large and complicated for our taste and there are millions of lines of C++ code ''out there'' that we endeavor not to break. All changes to the language must undergo

tremendous consideration. Additions to it are undertaken with great trepidation. Wherever possible we prefer to see programming techniques and library functions used as alternatives to language extensions.

Many communities of programmers want to see their favorite language construct or library class propagated into C++. Unfortunately, adding useful features from diverse communities could turn C++ into a set of incoherent features. C++ is not perfect, but adding features could easily make it worse instead of better.''

So, given that, what features have caused trouble by their absence and which are under debate so that they might make it into C++ over the next couple of years? The feature I regret not having put in much earlier when it could be done without formal debate and experimentation was easy is some form of name space control†. C++ follows C in having a single global name space. C++ alleviates the problems that causes with class scopes and overloading mechanisms, but as programs grow and especially as independent libraries are developed and later used together in a single program the problem of name space clashes increases. My only defense is that I didn't like the resolution mechanisms I looked at, such as Ada packages and Modula-2 modules, because they had too great overlap with the C++ class mechanism and thus didn't fit.

In the original C++ design, I deliberately didn't include the Simula mechanisms for run-time type identification (`QUA` and `INSPECT`). My experience was that they were almost always misused, so that the benefits from having the mechanism would be outweighed by the disadvantages. Several proposals for providing some form of run-time type identification have arisen over the years and the focus is now on a proposal for dynamic casts – that is, type conversions that are checked at run time – by Dmitry Lenkov and me [Stroustrup,1992a]. Like other extension proposals it is going through extensive discussion and experimentation and is unlikely to be accepted in the form presented in that paper†.

7.3 What Was The Biggest Mistake?

To my mind there really is only one contender for the title of ''worst mistake.'' Release 1.0 and my first edition [Stroustrup,1986b] should have been delayed until a larger library including some simple classes such as singly and doubly linked lists, an associative array class, a range checked array class, and a simple string class could have been included. The absence of those led to everybody re-inventing the wheel and to an unnecessary diversity in the most fundamental classes. However, could I have done that? In a sense, I obviously could. The original plan for my book included three library chapters, one on the stream library, one on the container classes, and one on the task library, so I knew roughly what I wanted. Unfortunately, I was too tired and couldn't do container classes without some form of templates.

7.4 Hopes for the Future

May C++ serve its user community well. For that to happen, the language itself must be stable and well-specified. The C++ standards group and the C++ compiler vendors have a great responsibility here.

In addition to the language itself, we need libraries. Actually, we need a libraries industry to produce, distribute, maintain, and educate people. This is emerging. The challenge is to allow programs to be composed out of libraries from different vendors. This is hard and might need some support from the standards committee in the form of standard classes and mechanisms that ease the use of independently developed libraries†.

---

† A facility for defining and using name spaces was introduced into C++ in 1994; see [Stroustrup,1994].

† After extensive discussion and revision, a mechanism for run-time type information based on this proposal was accepted in 1993; see [Stroustrup,1994].

† The standard library now provides containers and algorithms serving this need.

The language itself plus the libraries define the language that a user de facto writes in. However, only through good understanding of the application areas and design techniques will the language and library features be put to good use. Thus there must be an emphasis on teaching people effective design techniques and good programming practices. Many of the techniques we need have still to be developed and many of the best techniques we do have still compete with plain ignorance and snake oil. I hope for far better textbooks for the C++ language and for programming and design techniques, and especially for textbooks that emphasize the connection between language features, good programming practices, and good design approaches.

Techniques, languages, and libraries must be supported by tools. The days of C++ programming supported by simply a ''naked'' compiler are almost over and the best C++ tools and environments are beginning to approach the power and convenience of the best tools and environments for any language. We can do much better. The best has yet to come.

8 Acknowledgements

It is clear that given the number of people who have contributed to C++ in some form or other over the last 12 years or so, I must have left most unmentioned. Some have been mentioned in this paper, more can be found in §1.2c of the ARM, and other acknowledgement sections of my books and papers. Here I'll just mention Doug McIlroy, Brian Kernighan, Andrew Koenig, and Jonathan Shopiro who have provided constant help, ideas, and encouragement to me and others for a decade.

I'd also like to mention the people who have done much of the hard, but usually unnoticed and unacknowledged work of developing and supporting Cfront through the years when we were always short of resources: Steve Dewhurst, Laura Eaves, Andrew Koenig, Stan Lippman, George Logothetis, Glen McClusky, Judy Ward, Nancy Wilkinson, and Barbara Moo who managed AT&T Bell Lab's C++ development and support group during the years when the work got done.

Also thanks to C++ compiler writers, library writers, etc. who sent me information about their compilers and tools – most of which went beyond what could be presented in a paper: Mike Ball, Walter Bright, Keith Gorlen, Steve Johnson, Kim Knuttilla, Archie Lachner, Doug Lea, Mark Linton, Aron Insinga, Doug Lea, Dmitri Lenkov, Jerry Schwarz, Michael Tiemann, and Mike Vilot.

Also thanks to people who read drafts of this paper and provided many constructive comments: Dag Brück, Steve Buroff, Peter Juhl, Brian Kernighan, Andrew Koenig, Stan Lippmann, Barbara Moo, Jerry Schwarz, and Jonathan Shopiro.

Most comments on the various versions of this paper were of the form ''This paper is too long ... please add information about X, Y, and Z ... also be more detailed about A, B, and C.'' I have tried to follow the first part of that advice, though the constraints of the second part have ensured that this did not become a short paper. Without Brian Kernighan's help this paper would have been much longer.

9 References

[Babcisky,1984] Karel Babcisky: *Simula Performance Assessment.* Proc. IFIP WG2.4 conference on System Implementation Languages: Experience and Assessment. Canterbury, Kent, UK. September 1984.

[Birtwistle,1979] Graham Birtwistle, Ole-Johan Dahl, Bjørn Myrhaug, and Kristen Nygaard: *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden. 1979. ISBN 91-44-06212-5.

[Booch,1991] Grady Booch: *Object-Oriented Design*. Benjamin Cummings. 1991. ISBN 0-8053-0091-0.

[Booch,1990] Grady Booch and Michael M. Vilot: *The Design of the C++ Booch*

Components. Proc. OOPSLA'90.

[Call,1987] Lisa A. Call, et al.: – *An Open System for Graphical User Interfaces*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.

[Campbell,1987] Roy Campbell, et. al.: *The Design of a Multiprocessor Operating System*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.

[Cargill,1991] Tom A. Cargill: *The Case Against Multiple Inheritance in C++*. USENIX Computer Systems Vol 4, no 1,1991.

[Carroll,1991] Martin Carroll: *Using Multiple Inheritance to Implement Abstract Data Types*. The C++ Report. April 1991.

[Cristian,1989] Flaviu Cristian: Exception Handling. in *Dependability of Resilient Computers*, T. Andersen Editor, BSP Professional Books, Blackwell Scientific Publications, 1989.

[Cox,1986] Brad Cox: *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley. Reading, Massachusetts. 1986.

[Dahl,1988] Ole-Johan Dahl: Personal communication.

[Ellis,1990] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Massachusetts. 1990. ISBN 0-201-51459-1.

[Goldberg,1983] Adele Goldberg and David Robson: *Smalltalk-80, The language and its Implementation*. Addison-Wesley. 1983. ISBN 0-201-11371-6.

[Goodenough,1975] John Goodenough: *Exception Handling: Issues and a Proposed Notation*. CACM December 1975.

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

[Gorlen,1990] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico: *Data Abstraction and Object-Oriented Programming in C++*. Wiley. West Sussex. England. 1990. ISBN 0-471-92346-X.

[Ichbiah,1979] Jean D. Ichbiah, et.al.: *Rationale for the Design of the ADA Programming Language*. Sigplan Notices Vol 14, no 6, June 1979 Part B.

[Johnson,1989] Ralph E. Johnson: *The Importance of Being Abstract*. The C++ Report, Vol 1 No 3, March 1989.

[Kernighan,1981] Brian Kernighan and Dennis Ritchie: *The C Programming Language*. Prentice-Hall 1978. ISBN 0-13-110163-3.

[Kernighan,1981] Brian Kernighan: *Why Pascal is not my Favorite Programming Language*. AT&T Bell Labs Computer Science Technical Report No. 100. July 1981.

[Kernighan,1988] Brian Kernighan and Dennis Ritchie: *The C Programming Language (second edition)*. Prentice-Hall 1988. ISBN 0-13-110362-8.

[Kiczales,1992] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow: *The Art of the Metaobject Protocol*. The MIT Press. Cambridge, Massachusetts. 1991. ISBN 0-262-11158-6.

[Koenig,1988] Andrew Koenig: *Associative arrays in C++*. Proc. USENIX Conference. San Francisco. June 1988.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible – but no closer*. The C++ Report. Vol.1 No.7. July 1989.

[Koenig,1989b] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++*. Proc. ''C++ at Work'' Conference. November 1989.

[Koenig,1990] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++ (revised)*. Proc USENIX C++ Conference, April 1990. Also, Journal of Object Oriented Programming, Vol.3 No.2. July/Aug 1990. pp. 16-33.

[Krogdahl,1984] Stein Krogdahl: *An Efficient Implementation of Simula Classes with*

Multiple Prefixing. Research Report No. 83 June 1984, University of Oslo, Institute of Informatics.

[Lenkov,1989] Dmitry Lenkov: *C++ Standardization Proposal*. #X3J11/89-016.

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

[Liskov,1987] Barbara Liskov: *Data Abstraction and Hierarchy*. Addendum to Proceedings of OOPSLA'87. October 1987.

[Meyer,1988] Bertrand Meyer: *Object-Oriented Software Construction*. Prentice-Hall. 1988. ISBN 0-13-629049.

[McCluskey,1992] Glen McCluskey: *An Environment for Template Instantiation*. The C++ Report. February 1992.

[Nelson,1991] Nelson, G. (editor): *Systems Programming with Modula-3*. Prentice-Hall. 1991. ISBN 0-13-590464-1.

[Rose,1984] 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.

[Sakkinen,1992] Markku Sakkinen: *A Critique of the Inheritance Principles of C++*. USENIX Computer Systems, vol 5, no 1, Winter 1992.

[Sethi,1980] Ravi Sethi: A case study in specifying the semantics of a programming language. Seventh Annual ACM Symposium on Principles of Programming Languages. January 1980. pp 117-130.

[Shopiro,1985] Jonathan E. Shopiro: *Strings and lists for C++*. AT&T Bell Labs Internal Technical Memorandum. July 1985.

[Shopiro,1987] Jonathan E. Shopiro: Extending the C++ Task System for Real-Time Control. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.

[Snyder,1986] Alan Snyder: *Encapsulation and Inheritance in Object-Oriented Programming Languages*. Proc. OOPSLA'86. September 1986.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Labs Techinical Report HPL-94-34 (R. 1). August, 1994.

[Stroustrup,1978] Bjarne Stroustrup: *On Unifying Module Interfaces*. ACM Operating Systems Review Vol.12 No.1. pp 90-98. January 1978.

[Stroustrup,1979a] Bjarne Stroustrup: *Communication and Control in Distributed Computer Systems*. Ph.D. thesis, Cambridge University, 1979.

[Stroustrup,1979b] Bjarne Stroustrup: *An Inter-Module Communication System for a Distributed Computer System*. Proc. 1st Int'l Conf. on Distributed Computing Systems. October 1979. pp 412-418.

[Stroustrup,1980a] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. Bell Laboratories Computer Science Technical Report CSTR-84. April 1980. Also Sigplan Notices January, 1982.

[Stroustrup,1980b] Bjarne Stroustrup: *A Set of C Classes for Co-routine Style Programming*. Bell Laboratories Computer Science Technical Report CSTR-90. November 1980.

[Stroustrup,1981a] Bjarne Stroustrup: *Long Return: A technique for Improving The Efficiency of Inter-Module Communication*. Software Practice and Experience. January 1981. pp 131-143.

[Stroustrup,1981b] Bjarne Stroustrup: *Extensions of the C Language Type Concept* Bell Labs Internal Memorandum. January 1981.

[Stroustrup,1982] Bjarne Stroustrup: *Adding Classes to C: An Exercise in Language Evolution*. Bell Laboratories Computer Science internal document. April 1982.

Software Practice & Experience, Vol.13. 1983. pp. 139–61.

[Stroustrup,1984a] Bjarne Stroustrup: *The C++ Reference Manual*. AT&T Bell Labs Computer Science Technical Report No. 108 January 1984. (Written in the summer of 1983). Revised version November 1984.

[Stroustrup,1984b] Bjarne Stroustrup: *Data Abstraction in C*. Bell Labs technical Journal. Vol 63, No. 8. October 1984. pp 1701–1732. (Written in the summer of 1983)

[Stroustrup,1984c] Bjarne Stroustrup: *Operator Overloading in C++*. Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience & Assessment September 1984.

[Stroustrup,1985] Bjarne Stroustrup: *An Extensible I/O Facility for C++* Proc. Summer 1985 USENIX Conference. June 1985. pp 57–70.

[Stroustrup,1986a] Bjarne Stroustrup *An Overview of C++*. ACM Sigplan Notices, Special Issue pp 7–18 October, 1986.

[Stroustrup,1986b] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley. 1986. ISBN 0-201-12078-X.

[Stroustrup,1986c] Bjarne Stroustrup: *What is Object-Oriented Programming?* Proc. 14th ASU Conference. pp 69–84. August, 1986. Revised version in Proc. ECOOP'87, May 1987, Springer Verlag Lecture Notes in Computer Science Vol 276, pp 51–70. Revised version in *IEEE Software Magazine*, May 1988, pp 10–20.

[Stroustrup,1987a] Bjarne Stroustrup: *Multiple Inheritance for C++*. Proc. EUUG Spring Conference, May 1987. Also, USENIX Computer Systems, Vol 2 No 4, Fall 1989.

[Stroustrup,1987b] Bjarne Stroustrup and Jonathan Shopiro: *A Set of C classes for Co-Routine Style Programming.* Proc. USENIX C++ conference, Santa Fe. November 1987. pp 417–439.

[Stroustrup,1987c] Bjarne Stroustrup: *The Evolution of C++: 1985-1987*. Proc. USENIX C++ conference, Santa Fe. November 1987. pp 1–22.

[Stroustrup,1988a] Bjarne Stroustrup: *Type-safe Linkage for C++*. USENIX Computer Systems, Vol.1 No.4. Fall 1988.

[Stroustrup,1988b] Bjarne Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference, Denver. October 1988. pp. 1–18. Also, USENIX Computer Systems, Vol.2 No.1. Winter 1989.

[Stroustrup,1989a] Bjarne Stroustrup: *Standardizing C++*. The C++ Report. Vol.1 No.1. January 1989.

[Stroustrup,1989b] Bjarne Stroustrup: *The Evolution of C++: 1985-1989*. USENIX Computer Systems, Vol.2 No.3. Summer 1989. Revised version of [Stroustrup,1987c].

[Stroustrup,1990] Bjarne Stroustrup: *On Language Wars*. Hotline on Object-Oriented Technology. Vol 1, No 3. January 1990.

[Stroustrup,1991] Bjarne Stroustrup: *The C++ Programming Language (2nd edition)*. Addison-Wesley. 1991. ISBN 0-201-53992-6.

[Stroustrup,1992a] Bjarne Stroustrup and Dmitri Lenkov: *Run-Time Type Identification for C++*. The C++ Report. March 1992.

[Stroustrup,1992b] Bjarne Stroustrup: *How to Write a C++ Language Extension Proposal*. The C++ Report. May 1992.

[Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994. ISBN 0-201-54330-3.

[Taft,1992] S. Tucker Taft: *Ada 9X: A Technical Summary*. CACM. November 1992.

[Tiemann,1987] Michael Tiemann: *Wrappers* Proc. USENIX C++ conference, Santa Fe.

November 1987.

[Waldo,1991] Jim Waldo: *Controversy: The Case for Multiple Inheritance in C++*. USENIX Computer Systems, vol 4, no 2, Spring 1991.

[Wirth,1982] Niclaus Wirth: *Programming in Modula-2* Springer-Verlag. 1982.