0

Notes to the Reader

"The time has come," the Walrus said,
"to talk of many things."
– L.Carroll

This chapter consists of an overview of this book, a list of references, and some ancillary notes on C++. The notes concern the history of C++, ideas that influenced the design of C++, and thoughts about programming in C++. This chapter is not an introduction: the notes are not a prerequisite for understanding the following chapters, and some notes assume knowledge of C++.

The Structure of This Book

This book has three parts: Chapters 1 through 10 provide a tutorial introduction to C++. Chapters 11 through 13 present a discussion of design and software development issues arising in connection with the use of C++. Finally, the complete C++ reference manual is included. Complete information about language features can be found only in the reference manual. However, the tutorial chapters provide examples, opinions, recommendations, warnings, and exercises that have no place in a manual.

The emphasis of this book is on program organization rather than on the writing of algorithms. Consequently, where there was a choice, a trivial algorithm was chosen over a clever and harder-to-understand algorithm. For example, a bubble sort is used where a quicksort would be more appropriate for real code. Often, re-implementation with a more suitable algorithm is left as an exercise.

Chapter 1 is a quick tour of the major concepts and features of the C++ programming language. Its purpose is to give a high-level acquaintance with C++. Detailed explanations of language features and techniques are postponed to later chapters. The discussion focuses on the language features supporting data abstraction and object-

oriented programming, but also briefly introduces the key features used for procedural programming.

Chapters 2, 3, and 4 describe features of C++ that are not involved in defining new types: the fundamental types, expressions, and control structures for C++ programs. In other words, they describe the subset of C++ that is essentially C. They go into considerably greater detail than Chapter 1.

Chapters 5 through 8 describe C++'s facilities for defining new types, features that do not have counterparts in C. Chapter 5 presents the basic class concept, showing how user-defined types (classes) can be defined and how objects of such types can be initialized, accessed, and finally cleaned up. Chapter 6 describes the concept of a derived class, which enables a programmer to build more complex classes from simpler ones and to handle objects in an efficient and type-safe manner in contexts in which their type cannot be known at compile time. Chapter 7 explains how to define unary and binary operators for a user-defined type, how to specify conversions between user-defined types, and how to specify creation, deletion, and copying of a value of a user-defined type. Chapter 8 presents templates, that is, C++'s facilities for defining families of types and functions. It demonstrates their use in the definition of container classes such as lists and associative arrays.

Chapter 9 presents exception handling and discusses ideas for error handling and strategies for fault tolerance. Chapter 10 presents the `ostream` and `istream` classes provided for input and output in the *standard library*.

Chapters 11 through 13 discuss issues that arise when C++ is used in the design and implementation of large software systems. Chapter 11 concentrates on design and management issues, Chapter 12 discusses the relation between the C++ programming language and design issues; and Chapter 13 presents library design techniques.

Finally, the C++ reference manual is included.

References to parts of this book are of the form §2.3.4 (Chapter 2, sub-section 3.4). Chapter *r* is the reference manual; for example, §r.8.5.5.

Implementation Notes

Several independently developed and distributed implementations of C++ exist. A wealth of tools, libraries, and software development environments are also available. A mass of text books, manuals, journals, newsletters, electronic bulletin boards, mailing lists, conferences, and courses is available to inform you about the latest developments in C++, its use, details, tools, libraries, implementations, etc. If you plan to use C++ seriously, I strongly suggest that you gain access to at least two such sources – each source has a bias, so use at least two.

The major program fragments in this book were taken directly from source files that were compiled on a DEC VAX11/8550 running the 10th Edition UNIX[25] system. The compiler used is a direct descendant of the original C++ compiler developed by the author. The language used here is ''pure C++;'' that is, no implementation-dependent extensions are used. Therefore, the examples ought to run on any C++ implementation. However, the template and exception handling facilities are at the time of writing recent additions to C++, so your compiler may not yet support them.

Exercises

Exercises can be found at the end of chapters. The exercises are mainly of the write-a-program variety. Always write enough code for a solution to be compiled and run with at least a few test cases. The exercises vary considerably in difficulty, so they are marked with an estimate of their difficulty. The scale is exponential so that if a (∗1) exercise takes you five minutes, a (∗2) might take an hour, and a (∗3) might take a day. The time needed to write and test a program depends more on the reader's experience than on the exercise itself. A (∗1) exercise might take a day if the reader first has to get acquainted with a new computer system to run it. On the other hand, a (∗5) exercise might be done in an hour by someone who happens to have the right collection of programs handy.

Any book on programming in C can be used as a source of extra exercises for Chapters 2 through 4. Aho et al.[1] present many common data structures and algorithms in terms of abstract data types. It can therefore be used as a source of exercises for Chapters 5 through 8. However, the language used in that book lacks both member functions and derived classes. Consequently, the user-defined types can often be expressed more elegantly in C++.

Design Notes

Simplicity was an important design criterion for C++; where there was a choice between simplifying the manual and other documentation or simplifying the compiler, the former was chosen. Great importance was also attached to retaining compatibility with C; this precluded cleaning up C syntax.

C++ has no high-level data types and no high-level primitive operations. For example, there is no matrix type with an inversion operator or a string type with a concatenation operator. If a user wants such a type, it can be defined in the language itself. In fact, defining a new general-purpose or application-specific type is the most fundamental programming activity in C++. A well designed user-defined type differs from a built-in type only in the way it is defined and not in the way it is used.

Features that would incur run-time or memory overheads even when not used were avoided. For example, ideas that would make it necessary to store ''housekeeping information'' in every object were rejected; if a user declares a structure consisting of two 16-bit quantities, that structure will fit into a 32-bit register.

C++ was designed to be used in a traditional compilation and run-time environment, the C programming environment on the UNIX system. There are, however, good reasons for using C++ in an environment with significantly more support available. Facilities such as dynamic loading, incremental compilation, and a database of type definitions can be put to good use without affecting the language.

C++ types and data-hiding features rely on compile-time analysis of programs to prevent accidental corruption of data. They do not provide secrecy or protection against someone deliberately breaking the rules. They can, however, be used freely without incurring run-time or space overheads. The idea is that to be useful, a language feature must not only be elegant, it must also be affordable in the context of a real program.

Historical Note

Clearly C++ owes most to C[8]. C is retained as a subset, and so is C's emphasis on facilities that are low-level enough to cope with the most demanding systems programming tasks. C in turn owes much to its predecessor BCPL[13]; in fact, BCPL's `//` comment convention has been (re)introduced in C++. If you know BCPL you will notice that C++ still lacks a VALOF block. The other main source of inspiration was Simula67[2,3]; the class concept (with derived classes and virtual functions) was borrowed from it. The Simula67 `inspect` statement was deliberately not introduced into C++. The reason for that is to encourage modularity through the use of virtual functions. C++'s facility for overloading operators and the freedom to place a declaration wherever a statement can occur resembles Algol68[24].

Since the original edition of this book, the language has been extensively reviewed and refined. The major areas for revision were overloading resolution, linking, and memory management facilities. In addition, several minor changes were made to increase C compatibility. Several generalizations and a few major extensions were added: these include multiple inheritance, `static` member functions, `const` member functions, `protected` members, templates, and exception handling. The overall theme of these extensions and revisions was to make C++ a better language for writing and using libraries. These changes are described in references 10, 18, 20, 21, and 23.

The template facility was partly designed to formalize macro usage, partly inspired by the Ada generics (both their strengths and their weaknesses), and partly inspired by Clu's parameterized modules. Similarly, the C++ exception handling mechanism was partly inspired partly by Ada, by Clu[11], and partly by ML[26]. Other developments in the 1985 to 1991 time span – such as multiple inheritance, `static` member functions, and pure virtual functions – are generalizations driven by experience with the use of C++ rather than ideas imported from other languages.

Earlier versions of the language, collectively known as ''C with Classes''[16], have been in use since 1980. The language was originally invented because the author wanted to write some event-driven simulations for which Simula67 would have been ideal, except for efficiency considerations. ''C with Classes'' was used for major simulation projects in which the facilities for writing programs that use (only) minimal time and space were severely tested. ''C with Classes'' lacked operator overloading, references, virtual functions, and many details. C++ was first installed outside the author's research group in July, 1983; quite a few current C++ features had not yet been invented, however.

The name C++ (pronounced C plus plus) was coined by Rick Mascitti in the summer of 1983. The name signifies the evolutionary nature of the changes from C. ''++'' is the C increment operator. The slightly shorter name C+ is a syntax error; it has also been used as the name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C. The language is not called D, because it is an extension of C and does not attempt to remedy problems by removing features. For yet another interpretation of the name C++, see the appendix of Orwell[12].

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. Until recently, there never was a C++ paper design; design, documentation, and implementation went on simultaneously. There never was a ''C++ project'' either, or a ''C++ design committee.'' Throughout, C++ evolved, to cope with problems encountered by users, and through discussions between the author and his friends and colleagues.

Lately, the explosive growth of C++ use has caused some changes. Sometime during 1987, it became clear that formal standardization of C++ was inevitable and that we needed to start preparing the ground immediately for a standardization effort[22]. The result was a conscious effort to maintain contact between C++ compiler implementers and major users through paper and electronic mail and through face-to-face meetings at C++ conferences and elsewhere.

AT&T Bell Laboratories made a major contribution to this by allowing me to share drafts of revised versions of the C++ reference manual with such implementers and users. Because many of these people work for companies that could be seen as competing with AT&T, the significance of this contribution should not be underestimated. A less enlightened company could have caused major problems of language fragmentation simply by doing nothing. As it happened, about a hundred individuals from a couple of dozen organizations read and commented on what became the current reference manual and the base document for the ANSI C++ standardization effort. Their names can be found in *The Annotated C++ Reference Manual*[4]. The reference manual proper is part of this book. Finally, the X3J16 committee of ANSI was convened in December of 1989 at the initiative of Hewlett-Packard. It is expected that the ANSI (American national) standardization of C++ will be part of the ISO (international) standardization effort for C++.

C++ evolved hand-in-hand with some of the key classes presented in this book. For example, I designed the `complex`, `vector`, and `stack` classes together with the operator overloading mechanisms. The string and list classes was developed by Jonathan Shopiro and me as part of the same effort. Jonathan's string and list classes were the first such classes to see extensive use as part of a library. The `task` library described in reference 19 and in exercise 13 in §6.8 was part of the first C with classes program ever written. I wrote it and its associated classes to support Simula-style simulations. The task library has been revised and reimplemented, notably by Jonathan Shopiro, and is still in extensive use. The stream library as described in the first edition of this book was designed and implemented by me. Jerry Schwarz transformed it into the iostreams library (§10) using (among other ideas) Andrew Koenig's manipulator technique (§10.4.2). The `Map` class (§8.8) was designed by Andrew Koenig. Similarly, Andrew wrote the `Pool` class (§13.10) to make a library version of the class-specific memory management technique (§5.5.6) I had devised. The development of the template facility was influenced by the `Vector`, `Map`, `Slist`, and `sort` templates presented in Chapter 8.

C and C++

C was chosen as the base language for C++ because (1) it is versatile, terse, and relatively low-level; (2) it is adequate for most system programming tasks; (3) it runs everywhere and on everything; and (4) it fits into the UNIX programming environment. C has its problems, but a language designed from scratch would have some too, and we know C's problems. Most importantly, working with C enabled ''C with Classes'' to be a useful (if awkward) tool within months of the first thought of adding Simula-like classes to C.

As C++ became more widely used, and as the facilities it provided over and above those of C became more significant, the question of whether to retain compatibility was raised again and again. Clearly some problems could be avoided if some of the C heritage was rejected (see, for example, Sethi[15]). This was not done because (1) there are millions of lines of C code that might benefit from C++, provided that a complete rewrite from C to C++ were unnecessary; (2) there are millions of lines of library functions and utility software code written in C that could be used from/on C++ programs provided C++ were link compatible with and syntactically very similar to C; (3) there are hundreds of thousands of programmers who know C and therefore need only learn to use the new features of C++ and not relearn the basics; and (4) because C++ and C will be used on the same systems by the same people for years, the differences should be either very large or very small to minimize mistakes and confusion. The definition of C++ has been revised to ensure that a construct that is both legal C and legal C++ actually has the same meaning in both languages.

The C language has itself evolved, partly under the influence of the development of C++ (see Rosler[14]). The ANSI C standard[27] contains a function declaration syntax borrowed from ''C with Classes.'' Borrowing works both ways; for example, the `void*` pointer type was invented for ANSI C and first implemented in C++. As promised in the first edition of this book, the definition of C++ has been reviewed to remove gratuitous incompatibilities; C++ is now more compatible with C than it originally was; see §r.18. The ideal is for C++ to be as close to ANSI C as possible – but no closer[9]. One hundred percent compatibility is not and never was a goal because that would compromise the type safety and smooth integration of user-defined and built-in types that are among the essential aims of C++.

Knowing C is not a prerequisite for learning C++. Programming in C encourages many techniques and tricks rendered unnecessary by C++ language features. For example, explicit type conversion, ''casting,'' is less frequently needed in C++ than it is in C (see also the ''Note to C programmers'' below). However, *good* C programs tend to be C++ programs. For example, note that every program in Kernighan and Ritchie: The C Programming Language (2nd Edition)[8] is a C++ program. Experience with any statically typed language will be a help with learning C++.

Efficiency and Structure

C++ was developed from the C programming language and, with few exceptions, retains C as a subset. The base language, the C subset of C++, is designed so that there is a very close correspondence between its types, operators, and statements and

the objects computers deal with directly: numbers, characters, and addresses. Except for the `new`, `delete`, and `throw` operators and the `try` block, individual C++ expressions and statements need no hidden run-time support or subroutines.

C++ can use the same function call and return sequences as C – or even more efficient ones. When even such relatively efficient mechanisms are too expensive, a C++ function can be substituted inline, thus enjoying the notational convenience of functions without run-time overhead.

One of the original aims for C was to replace assembly coding for the most demanding systems programming tasks. When C++ was designed, care was taken not to compromise the gains in this area. The difference between C and C++ is primarily in the degree of emphasis on types and structure. C is expressive and permissive. C++ is even more expressive, but to gain that increase in expressiveness, the programmer must pay more attention to the types of objects. Knowing the types of objects, the compiler can deal correctly with expressions when the programmer would otherwise have had to specify operations in painful detail. Knowing the types of objects also enables the compiler to detect errors that would otherwise have persisted until testing. Note that using the type system to get function argument checking, to protect data from accidental corruption, to provide new types, to provide new operators, etc., does not in itself increase run-time or space overheads.

The emphasis on structure in the design of C++ reflects the increase in the scale of programs written since C was designed. You can make a small program (less than 1000 lines) work through brute force even when breaking every rule of good style. For a larger program, this is simply not so for an individual. If the structure of a 10,000 line program is bad, you will find that new errors are introduced as fast as old ones are removed. C++ was designed to enable larger programs to be structured in a rational way so that it would not be unreasonable for a single person to cope with 25,000 lines of code. C++ has by now been shown to over-fulfill that goal.

Much larger programs exist, but the ones that work generally turn out to consist of many nearly independent parts, each one well below the limits previously mentioned. Naturally, the difficulty of writing and maintaining a program depends on the complexity of the application and not simply on the number of lines of program text, so the exact numbers used to express the preceding ideas should not be taken too literally.

However, not every piece of code can be well structured, hardware independent, easy to read, etc. C++ possesses features that are intended for manipulating hardware facilities in a direct and efficient way without regard for safety or ease of comprehension. It also possesses facilities for hiding such code behind elegant and safe interfaces.

Naturally, the use of C++ for larger programs leads to the use of C++ by groups of programmers. C++'s emphasis on modularity, strongly typed interfaces, and flexibility pays off here. C++ has as good a balance of facilities for writing *large* programs as any language. However, as programs get larger, the problems associated with their development and maintenance shift from being language problems to more global problems of tools and management. Chapters 11 and 12 explore some of these issues.

This book emphasizes techniques for providing general-purpose facilities, generally useful types, libraries, etc. These techniques will serve programmers of small programs as well as programmers of large ones. Furthermore, because all non-trivial programs consist of many semi-independent parts, the techniques for writing such parts serve programmers of both systems and applications.

One might suspect that specifying a program using a more detailed type structure would lead to a larger program source text. With C++ this is not so; a C++ program declaring functions argument types, using classes, etc., is typically a bit shorter than the equivalent C program not using these facilities. Also, where libraries are used a C++ program will *appear* much shorter than its C equivalent, assuming, of course, that a functioning C equivalent could actually have been built.

Philosophical Note

A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed, and it provides 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.

The connection between the language in which we think/program and the problems and solutions we can imagine is very close. For this reason, restricting language features with the intent of eliminating programmer errors is at best dangerous. As with natural languages, there are great benefits from being at least bilingual. The language provides a programmer with a set of conceptual tools; if these are inadequate for a task, they will simply be ignored. For example, seriously restricting the concept of a pointer simply forces the programmer to use a vector plus integer arithmetic to implement structures, pointers, etc. Good design and the absence of errors cannot be guaranteed merely by the presence or the absence of specific language features.

The type system should be especially helpful for non-trivial tasks. The C++ class concept has, in fact, proven itself to be a powerful conceptual tool.

Thinking about Programming in C++

Ideally, one approaches the task of designing a program in three stages: first gain a clear understanding of the problem, then identify the key concepts involved in a solution, and finally express that solution in a program. However, the details of the problem and the concepts of the solution often become clearly understood only through the effort to express them in the program – this is where the choice of programming language matters.

In most applications there are concepts that are not easily represented in a program as one of the fundamental types or as a function without associated static data. Given such a concept, declare a class to represent it in the program. A class is a type; that is,

it specifies how objects of its class behave: how they are created, how they can be manipulated, and how they are destroyed. A class also specifies how objects are represented, but at the early stages of the design of a program, that is not (and should not be) the major concern. The key to writing a good program is to design classes so that each cleanly represents a single concept. Often this means that the programmer must focus on the questions: How are objects of this class created? Can objects of this class be copied and/or destroyed? What operations can be done on such objects? If there are no good answers to such questions, the concept probably wasn't ''clean'' in the first place, and it might be a good idea to think a bit more about the problem and the proposed solution instead of immediately starting to ''code around'' the problems.

The concepts that are easiest to deal with are the ones that have a traditional mathematical formalism: numbers of all sorts, sets, geometric shapes, etc. There really ought to be standard libraries of classes representing such concepts, but this is not the case at the time of writing. There is a bewildering wealth of such libraries but as yet no formal or de facto standard. C++ is still young, and its libraries have not yet matured to the same degree as the language itself.

A concept does not exist in a vacuum; there are always clusters of related concepts. Organizing the relationship between classes in a program – that is, determining the exact relationship between the different concepts involved in a solution – is often harder than laying out the individual classes in the first place. The result had better not be a muddle in which every class (concept) depends on every other. Consider two classes, A and B: Relationships such as ''A calls functions from B,'' ''A creates Bs,'' and ''A has a B member'' seldom cause major problems. Relationships such as ''A uses data from B'' can typically be eliminated.

One of the most powerful intellectual tools for managing complexity is hierarchical ordering, that is, organizing related concepts into a tree structure with the most general concept at the root. In C++, derived classes represent such structures. A program can often be organized as a set of trees or directed acyclic graphs of classes. That is, the programmer specifies a number of base classes, each with its own set of derived classes. Virtual functions (§6.2.5) can often be used to define a set of operations for the most general version of a concept (a base class). When necessary, the interpretation of these operations can be refined for particular special cases (derived classes).

Naturally, this organization has its limits. Sometimes even a directed acyclic graph seems insufficient for organizing the concepts of a program; some concepts seem to be inherently mutually dependent. If a set of mutually dependent classes is so small that it is easy to understand, the cyclic dependencies need not be a problem. The idea of `friend` classes (§5.4.1) can be used to represent sets of mutually dependent classes in C++.

If you can organize the concepts of a program only into a general graph (and not a tree or a directed acyclic graph), and if you cannot localize the mutual dependencies, then you are most likely in a predicament that no programming language can help you out of. Unless you can conceive of some easily stated relationships between the basic concepts, the program is likely to become unmanageable.

Another form of commonality can be expressed through templates. A class template specifies a family of classes. For example, a list template specifies ‘‘list of T’’ where ‘‘T’’ can be any type. Thus a template is a mechanism for specifying how one type is generated given another type as an argument. The most common templates are container classes such as lists, arrays, and associative arrays.

Remember that much programming can be simply and clearly done using only primitive types, data structures, plain functions, and a few classes from a standard library. The whole apparatus involved in defining new types should not be used except when there is a real need.

The question ‘‘How does one write good programs in C++?’’ is very similar to the question ‘‘How does one write good English prose?’’ There are two kinds of answers: ‘‘Know what you want to say,’’ and ‘‘Practice. Imitate good writing.’’ Both kinds of advice appear to be as appropriate for C++ as they are for English – and as hard to follow.

Rules of Thumb

Here is a set of ‘‘rules’’ you might consider while learning C++. As you get more proficient you can evolve them into something suitable for your kind of applications and your style of programming. They are deliberately very simple, so they lack detail. Don’t take them too literally. To write a good program takes intelligence, taste, and patience. You are not going to get it right the first time; experiment!

[1] When you program, you create a concrete representation of the ideas in your solution to some problem. Let the structure of the program reflect those ideas as directly as possible:
  [a] If you can think of ‘‘it’’ as a separate idea, make it a class.
  [b] If you can think of ‘‘it’’ as a separate entity, make it an object of some class.
  [c] If two classes have something significant in common, make that commonality a base class.
  [d] If a class is a container of objects, make it a template.

[2] When you define a class that does not implement a mathematical entity like a matrix or a complex number, or a low-level type such as a linked list:
  [a] Don’t use global data.
  [b] Don’t use global (nonmember) functions.
  [c] Don’t use public data members.
  [d] Don’t use friends, except to avoid [a], [b], or [c].
  [e] Don’t access data members of another object directly.
  [f] Don’t put a ‘‘type field’’ in a class; use virtual functions.
  [g] Don’t use inline functions, except as a significant optimization.

Note to C Programmers

The better one knows C, the harder it seems to avoid writing C++ in C style, thereby losing some of the potential benefits of C++. So please take a look at the ‘‘Differences from C’’ section of the reference manual (§r.18). Here are a few pointers to the

areas in which C++ has better ways of doing something than C has. Macros are almost never necessary in C++; use `const` (§2.5) or `enum` (§2.5.1) to define manifest constants, `inline` (§4.6.2) to avoid function-calling overhead, and `templates` (§8) to specify families of functions and types. Don't declare a variable before you need it so that you can initialize it immediately – a declaration can occur anywhere a statement can. Don't use `malloc()` – the `new` operator (§3.2.6) does the same job better. Unions are needed less often than in C because derived classes often provide an alternative; try doing without them and when you do need a union keep it out of major interfaces – try anonymous unions (§2.6.2). Try to avoid `void*`, pointer arithmetic, C arrays, and casts except deep within the implementation of some function or class. Note that to obey C linkage conventions, a C++ function must be declared `extern "C"` (§4.4).

Most importantly, try thinking of a program as a set of interacting concepts represented as classes and objects, instead of the program as a bunch of data structures with functions twiddling their bits.

References

There are few direct references in the text, but here is a short list of books and papers that are mentioned directly or indirectly.

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman: *Data Structures and Algorithms.* Addison-Wesley, Reading, Massachusetts. 1983.

[2] O-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language.* Norwegian Computing Center S-22. Oslo, Norway. 1970.

[3] O-J. Dahl and C.A.R. Hoare: *Hierarchical Program Construction* in *Structured Programming.* Academic Press, New York. 1972. pp. 174-220.

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

[5] A. Goldberg and D. Robson: *SMALLTALK-80 – The Language and Its Implementation.* Addison-Wesley, Reading, Massachusetts. 1983.

[6] R.E. Griswold et.al.: *The Snobol4 Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey. 1970.

[7] R.E. Griswold and M.T. Griswold: *The ICON Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey. 1983.

[8] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey. 1978. Second edition 1988.

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

[10] 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.

[11] Barbara Liskov et al.: *CLU Reference Manual*. MIT/LCS/TR-225.

[12] George Orwell: *1984.* Secker and Warburg, London. 1949.

[13] Martin Richards and Colin Whitby-Strevens: *BCPL - The Language and Its Compiler*. Cambridge University Press. 1980.

[14] L. Rosler: *The Evolution of C - Past and Future.* AT&T Bell Laboratories Technical Journal. Vol.63 No.8 Part 2. October 1984. pp. 1685-1700.
[15] Ravi Sethi: *Uniform Syntax for Type Expressions and Declarations.* Software Practice & Experience, Vol.11. 1981. pp. 623-628.
[16] Bjarne Stroustrup: *Adding Classes to C: An Exercise in Language Evolution.* Software Practice & Experience, Vol.13. 1983. pp. 139-61.
[17] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley. 1986.
[18] Bjarne Stroustrup: *Multiple Inheritance for C++*. Proc. EUUG Spring Conference, May 1987. Also, USENIX Computer Systems, Vol 2 No 4, Fall 1989.
[19] 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.
[20] Bjarne Stroustrup: *Type-safe Linkage for C++*. USENIX Computer Systems, Vol.1 No.4. Fall 1988.
[21] 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.
[22] Bjarne Stroustrup: *Standardizing C++*. The C++ Report. Vol.1 No.1. January 1989.
[23] Bjarne Stroustrup: *The Evolution of C++: 1985-1989*. USENIX Computer Systems, Vol.2 No.3. Summer 1989.
[24] P.M. Woodward and S.G. Bond: *Algol 68-R Users Guide.* Her Majesty's Stationery Office, London. 1974.
[25] *UNIX Time-Sharing System: Programmer's Manual. Research Version, Tenth Edition.* AT&T Bell Laboratories, Murray Hill, New Jersey. February 1985.
[26] Aake Wilkstroem: *Functional Programming Using ML*. Prentice-Hall, Englewood Cliffs, New Jersey. 1987.
[27] X3 Secretariat: *Standard - The C Language.* X3J11/90-013. Computer and Business Equipment Manufactures Association, 311 First Street, NW, Suite 500, Washington, DC 20001, USA.

References to books relating to design and larger software development issues can be found at the end of Chapter 11.