JYVÄSKYLÄ STUDIES IN COMPUTER SCIENCE, ECONOMICS AND STATISTICS

20

Markku Sakkinen

Inheritance and Other Main Principles of C++ and Other Object-oriented Languages

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 1992

JYVÄSKYLÄ STUDIES IN COMPUTER SCIENCE, ECONOMICS AND STATISTICS 20

Markku Sakkinen

Inheritance and Other Main Principles of C++ and Other Object-oriented Languages

UNIVERSITY OF JYVÄSKYLÄ, JYVÄSKYLÄ 1992

ISBN 951-680-817-4 (nid.), 978-951-39-4352-3 (PDF)
ISSN 0357-9921

Jyväskylän yliopiston monistuskeskus and
Sisäsuomi Oy, Jyväskylä 1992

To Professor Ilppo Simo Louhivaara,
who for so long
tried to make me a mathematician

ABSTRACT

Sakkinen, Markku

Inheritance and Other Main Principles of C++ and Other Object-oriented Languages / Markku Sakkinen

Jyväskylä: University of Jyväskylä, 1992.

208 p.

(Jyväskylä Studies in Computer Science, Economics and Statistics
ISSN 0357-9921; 20)
ISBN 951-680-817-4 (nid.), 978-951-39-4352-3 (PDF)

Diss.

A definition of ‘object’, and consequently of ‘object-oriented programming language’, is presented, based in part on many earlier definitions from the literature. The partially object-oriented language C++ is assessed twice, first as it was in 1988, then in its current version. Some severe defects and a host of small faults are found. Mainly, its compatibility with C keeps it low-level and unsafe, but also its object orientation is insufficient. Because C++ is very popular and widely used today, all problems in it must be taken seriously. — The “Law of Demeter”, a style rule for object-oriented programming is analysed, and adapted better to the particularities of C++. Some changes and clarifications to the law in general are suggested. — Some important facets of inheritance are studied; a clear distinction between *essential* and *incidental* inheritance is made. Inheritance is modelled to a large part by composition. Multiple inheritance is divided into two varieties: independent and fork-join. It is claimed that the multiple-inheritance principles of most languages except C++ violate the integrity of superclasses in fork-join inheritance. A further analysis uncovers several anomalies also in the inheritance principles of C++, especially concerning multiple inheritance. Concrete corrections to all these problems are proposed. It is made plausible that multiple inheritance can be defined in a consistent and conceptually satisfying way. Implementation problems are not treated, however. — Lastly, several promising directions for continuing this research are briefly presented.

Key words and phrases: object orientation, pr ogramming languages, object-oriented programming, C++, inheritance, multiple inheritance, programming style.

ACM Computing Reviews categories:
D.3.2 Programming Languages: Language Classifications
Object-oriented languages
D.1.5 Programming Techniques: Object-oriented Programming
D.3.3 Programming Languages: Language Constructs and Features

Markku Sakkinen
Department of Computer Science and Information Systems
University of Jyväskylä
PL 35
SF-40351 Jyväskylä
Finland

Electronic mail: sakkinen@jytko.jyu.fi
or (EARN/BITNET:) SAKKINEN@FINJYU
Telephone: +358 41 603016
Telefax: +358 41 603611

PREFACE

When I returned to work for the University of Jyväskylä in August 1985, one of my incentives was to earn the doctor's degree in Computer Science before retirement age. Nine years earlier, I had earnestly tried to do doctoral studies in Mathematics, spurred by the principal professor of my studying years, Ilppo Simo Louhivaara, and also supported by professors Wolfgang Tutschke (Halle) and Heinrich Begehr (Berlin, Free University). That was in vain — I had already lost more than half of my heart to computers, and in programming tasks (if not already during military service, immediately after the Lic.Phil. degree in 1972) had also lost my grip on mathematics.

At the department of Computer Science[1], although my position as a laboratory engineer was in principle technical and practical instead of a research job, I was met with an atmosphere that encouraged all research interests. For a small department (now two full and two associate professors, and disproportionately few lower teaching and research positions), the spectrum of research topics is very wide. The most important person for me was associate professor Seppo Sippu, although I was not interested in his own speciality, which was parsing theory at that time (he has published an impressive two-volume book on it with Eljas Soisalon-Soininen).

The most directly interesting research at the department, from my viewpoint, was Airi Salminen's work on document databases which was partially supervised by Sippu. She studied several interesting models for documents and document databases before the one that was presented in her dissertation in 1989. I began cooperation with Salminen (and Sippu) in order to find some closely related topic that could serve for my own dissertation. Slowly, mainly during the year 1988, object-oriented programming grew to be the main area of my research efforts, which it still seems to remain at least for a couple of years.

It was evident that I would not be able to achieve much research beside my ordinary duties: despite the supportive atmosphere, there was so much to do. I also had three children, which limited the amount of evening and weekend time that I was willing to offer for science. Fortunately, there have been many periods of full-time research (at least in principle), for which I am very grateful.

[1] The English name has been later prolonged with 'and Information Systems', although the official name of the department in Finnish has not changed.

The first research period of 11 months in 1987 was in a project funded by the Academy of Finland and led by Seppo Sippu. It was immediately followed by fundings from the national doctoral programme in Computing Technology (6 months), another project of the Academy of Finland (6 months), led by Eljas Soisalon-Soininen (University of Helsinki), and from the national doctoral programme in Computer Science (7 months). Later, I was employed for the whole year 1991 by the project, ''Object-oriented languages and techniques'', funded again by the Academy of Finland and led by Kai Koskimies (University of Tampere). This project, which is continuing until the end of 1992, has been fruitful e.g. in connecting researchers from three universities.

Considering the total duration of the research periods, it has taken much longer to write this thesis than I had expected; it would have taken still longer in monograph form. Perhaps the main reason is that I always studied those problems that happened to interest me most, instead of taking guidance and following a laid-out plan. Now, finishing the thesis, I do not feel a threatening emptiness after a great achievement; rather there seems to be a large number of promising topics for further research.

I hope that all people who have had a direct influence on one or more of the constituent papers of this dissertation have been duly acknowledged in those papers. My thesis supervisor was Seppo Sippu until the end of 1990, when he moved to the University of Helsinki, and Kai Koskimies after that. Here is the place also to thank the official reviewers of the thesis, professors Jørgen Lindskov Knudsen (Aarhus University) and Karl Lieberherr (Northeastern University), for their thorough work and their suggestions. It is very pleasant that Dr. Oscar Nierstrasz (University of Geneva) has promised to act as my ''opponent'' (examiner).

There are several other people whose influence has been more indirect but still significant. For instance, many coworkers at Valmet Corporation and Procons OY in 1979 − 1985, and colleagues at the department after that (not least those two who take care of almost everything: amanuensis Mirja Tervo and department secretary Kaarina Suonia), as well as some people from the Computing Centre and from the Department of Mathematics. The international ''object-oriented fraternity'', especially the organisers and participants of the annual ECOOP conferences since 1988 (I did not yet attend the first ECOOP in 1987), has been very stimulating.

One important factor in my research has been the Internet, together with its user community. Its rapid evolution during the last few years has provided us with instant electronic mail to the most important countries, a wealth of newsgroups (actually *discussion* groups) on the most diverse topics, and the ability to fetch even large collections of files from many sites in many countries. Thanks to the net, I have been able both to get brand new results and papers from other researchers and to have some of my own ideas and drafts very quickly criticised. It would be impossible to list all people whose messages and newsgroup postings have been useful, while not always pleasant at the moment. Also to all those people, mostly ''invisible'', who work hard to keep the local, national and international networks operational and to enhance their services, I express my sincere gratitude.

Incidentally, I consider it unfortunate that English has become the international language of science in general and Computer Science in particular, for two different reasons. The first is that English is so baroque and unstructured, at least by the criteria on which *programming* languages are usually judged. The second reason is that the selection of *any* national language as the prevailing international language of science is unfair; Latin or Esperanto would be neutral. — This is not meant to be an excuse for any bad English in the dissertation. I have certainly tried to write as well as possible.

The text of this dissertation has been produced using Groff, the freeware (GNU) text formatter produced by James Clark and compatible with device-independent Troff, with the "-me" macro package. It was relatively easy to modify the chapters to a unified format, although they had been originally output in different styles and layouts.

Last but not least, I want to thank my family for the support and encouragement that they have given to my endeavours: my wife Ritva, my children Annukka, Päivikki and Jarkko, and my parents Kerttu and Lauri. Fortunately, the usual apologies for neglecting the family for the benefit of the dissertation are not due; they had often been much worse off when I was doing ''real work''. I can count on loyal family support even for my future scientific work.

Jyväskylä, July 1992
Markku Sakkinen

CONTENTS

CHAPTER 1

OVERVIEW AND OUTLOOK

OVERVIEW AND OUTLOOK

Structure of the thesis

The main contribution of this thesis consists of papers that either have been published or will appear in various conference proceedings, journals, and newsletters; reprinted here as Chapters 2 through 6. The year of first publication was 1988 for Chapters 2 and 3, 1989 for Chapter 4, and 1992 for Chapters 5 and 6.

I regarded it as important for readers that the constituent papers be really reprinted as integral parts of the dissertation. I am grateful to the Publication Board of the university and to the series editor, Airi Salminen, for allowing this, somewhat contrarily to currently recommended practice. Thanks are due as well to the original publishers for allowing reprinting or parallel publication, either as a standard policy or by granting a specific permission. The publishers are Springer-Verlag (2 and 6), Association for Computing Machinery (3), Cambridge University Press (4), and the University of California Press (5).

In order to keep the internal references consistent, chapter numbers have not been prepended to section and subsection numbers in the reprinted parts. For consistency, the chapter number has been left out also in this first chapter. This may cause some inconvenience to readers browsing to and fro within the whole book. Another inconvenience typical for "bundle" dissertations is that every chapter has its own list of references (and in a format slightly different from the others).

The rest of Chapter 1 is not necessarily best read contiguously. The subsections of Section 2 are most naturally read together with the corresponding chapter, and Sections 3 and 4 might as well have been organised as Chapter 7. The other chapters are ordered by the time of original publication, and need not be read in that order. Professor Knudsen indeed suggested tentatively that Chapter 2 could be omitted entirely from the thesis. However, the later chapters (6 in particular) refer to it for some observations that I did not want to repeat.

Chapter 5 requires much more previous knowledge of C++ than the other parts of the dissertation. Readers who are not very familiar with C++ previously should probably read Chapters 2 (at least its Appendix) and 6, in that order, before embarking on Chapter 5.

1. Definitions of object orientation

1.1. Introduction

'Object-oriented' (OO) has become such a fashionable term during the last few years that many of those people who really work with object-oriented systems and research are feeling uncomfortable with its indiscriminate use [King 89]. Nevertheless, it has a solid enough kernel of meaning to remain usable, especially to characterise programming languages and programming style. The current dissertation focusses on object-oriented programming *languages* (OOPLs), mainly sequential ones, rather than on object-oriented *programming* (OOP). I have taken into account some ideas and insights from object-oriented databases (OODBs). This is purely a matter of restriction, and in no way do I wish to denigrate the importance of the object-oriented approach in areas such as user interfaces, specification languages, information systems analysis and design, and so forth.

A "solid kernel" does not mean that there would be any single definition of the essential features of object orientation upon which a majority of the researchers could agree. Certainly a language's being object oriented or not is not a "yes or no" question: different people at least give different weights to the various criteria. There are, on the one hand, relatively few languages that everybody would label absolutely object oriented, and on the other hand, few modern high-level languages that have no object-oriented features whatsoever.

The book [Masini &c 91], which describes and compares a wide spectrum of more or less object-oriented languages, avoids giving a firm short definition of objects or object orientation. That looks reasonable in today's situation. It is more important to know what assumptions each author or subculture makes about object orientation, than what the "correct" meaning of the term is.

One thing that fascinates me is that one can study objects and object-oriented systems from many viewpoints and get rather different interpretations even for the basic features. A challenge then is to develop languages and models that are consistent from as many viewpoints as possible.

1.2. Some well-known characterisations

One rather popular definition, obviously originating from the Smalltalk community, is that the heart of object orientation is "the message-passing paradigm". I have mentioned my dissatisfaction with this notion in more than one of the constituent papers, but only in passing. Why do I really think that this phrase is out of place in the meaning of dynamically bound procedure invocations with an implicit **this** or **self** argument?

'Message passing' has an established meaning in computer communications and concurrent programming. The two main architectures of concurrent systems are tightly coupled systems, in which processes may communicate using shared storage, and loosely coupled systems, in which communication must happen by message passing [Atkinson 91 §6.1]. The following key characteristics of true message passing do not apply to Smalltalk-style "message passing":

- The communication may be asynchronous, the sender can continue its execution immediately after sending a message.
- The receiver can process messages one at a time (ordinary OOP requires a stack for active method invocations).
- The receiver may get to know the address or identity of the sender.
- The receiver may decide the order in which it processes queued messages, and may even discard some messages.
- Broadcast or group messages can be sent in some systems.

For instance, the ABCL/1 language, which is object-oriented and concurrent, has three message-passing types: past, now, and future [Yonezawa &c 87b §3]. There are also two modes, ordinary and express, where an express message can interrupt the processing of an ordinary one. "Now" messages are closest to Smalltalk messages, but they cannot be sent recursively [Yonezawa &c 87b §4], or deadlock will occur.

The following simple formula, mentioned e.g. in [Danforth &c 88] (and slight variants) seems to be quite popular as well:

object-oriented programming = abstract data types + inheritance

Obviously the possibility of dynamic binding is here implied by inheritance. This simplification goes too far at least because it does not say anything about the identity and mutability of objects.

In general, one should be a little cautious about the differences between abstract data types (ADTs) and objects. Foremost, ADT theory tends to be highly value-oriented. It may apply well to many types of small and simple objects, which have well-defined values. When we regard complex, application-oriented objects, it is impossible to define their *values* sensibly and unambiguously. Such an object cannot be treated as a black box, as many ADT approaches require: an operation on it may cause far-reaching side effects to many other objects, and vice versa. However, the same situation appears also in CLU, probably the best-known ADT language. See [Liskov 88] for a comparison of CLU and object-oriented programming.

A short characterisation of OOP according to the Scandinavian school appears e.g. in [Knudsen &c 88]:

A program execution is regarded as a physical model, simulating the behavior of either a real or imaginary part of the world.

This is very broad and conceptual, omitting all technical issues. It seems also that this definition suits best for the large application-oriented objects, and not so well for the more primitive objects which implement the former ones. It is thus somewhat complementary to the previous definition.

The single presentation that opens the widest perspective on the landscape of object-oriented programming is probably [Wegner 90]. However, it still maintains prominent a one-dimensional hierarchic classification, already presented in [Wegner 87]:

object-based languages: the class of all languages that support objects
class-based languages: the subclass that requires all objects to belong to a class
object-oriented languages: the subclass that requires classes to support inheritance

This classification has become rather popular in the literature, but it does not seem extremely relevant to me. For instance, those languages that are based on *prototypes* or *exemplars* instead of classes, can reach only the "object-based" level, no matter how many other object-oriented properties they have. (These languages are taken into account later in Wegner's papers, however.)

Even Wegner does not really point out the difference between objects and values. This has been done very extensively in [MacLennan

82]. Practically the only detail on which I cannot agree with MacLennan is the following claim (p. 77):

> Therefore, everything "in" a computer is an object; there are no values in computers.

My view is that both objects and values can be regarded without directly considering an implementation or execution. (An approach that tries to forbid treating pieces of software as abstract entities [Fetzer 88] is a serious fundamental misunderstanding.) When a programme is executed, both objects and values must be concretely represented in the computer: the former roughly as storage areas, the latter as bit patterns stored in them.

1.3. The Object-Oriented Database System Manifesto

One consensus-based list of the defining properties of object orientation is in [Atkinson &c 89]. The authors are all OO *database* researchers, but they divide their requirements into DBMS-related and OO-related ones; we omit the former here. The mandatory features presented in the Golden Rules of the Manifesto are the following (in italics), with my short comments.

1. Complex objects. Object-oriented programming languages mostly do not support these very well: there is no way to distinguish a part□whole relationship from mere associations between objects. Languages that are not restricted to reference semantics, e.g. C++, are better than most others in this respect.

2. Object identity. This is extremely important. Purely value-based approaches that omit the concept of object identity, such as functional (applicative) programming and relational databases, cannot be truly object oriented, no matter how many other features (typically inheritance) they adopt.

3. Encapsulation; information hiding is subsumed here, as is the custom in a large part of the literature.

4. Types or classes, where a type is regarded more as a compile-time, a class as a run-time notion. (This is in a way database-related as a mandatory feature. Programming languages can be based on the prototype approach, but a database without a schema does not look sensible).

5. Class or type hierarchies, i.e. inheritance. Four types of inheritance are listed: substitution, inclusion, constraint, and specialisation.

6. Overriding, overloading and late binding. This needs no comment.

7. Computational completeness. In the world of programming languages, this is already an implicit assumption.

8. Extensibility, in the meaning that new types can be defined and there is no distinction in usage between system-defined and user-defined types. In hybrid OO languages, system-defined types are often *weaker* than user-defined ones: e.g. they cannot be inherited from. (In extensible database systems the situation may often be the converse.)

There is a short list of optional features (or goodies) in [Atkinson &c 89]. *Multiple inheritance* is the only one among these that clearly belongs to object orientation. Further, at least *type checking and type inferencing* is tangential. A few things are labelled as open choices in the manifesto. All of these concern object orientation to some extent: *programming paradigm*, *representation system*, *type system*, and *uniformity*.

The Manifesto is not aimed to be the final word on the subject. Indeed, the last rule given at the very end is: "Thou shalt question the golden rules." (All rules are formulated in such archaic English.)

1.4. My own view

I still consider the list in Ch. 4 §2□3 quite adequate. If I try to condense my current view into a couple of short sentences, it becomes:

Definition: An object-oriented programming language supports the encapsulation of data and behaviour in objects with strong identity and integrity and some degree of information hiding. Normally it allows inheritance, with the dynamic binding of operations, either between objects or between object classes. Normally it also allows objects to be created, modified, and deleted.

Note that 'encapsulation' here is *not* meant to imply information hiding or data abstraction in itself.

What is meant by a strong notion of object identity is explained well in [Khoshafian &c 86]. However, the question becomes a little more complicated if complex objects in the sense of the previous subsection are supported, and if superclass subobjects are considered (Ch. 6 §3.8). The definition is very permissive in that it does not absolutely require mutable objects — although I cannot really imagine a sensible object-oriented language without them. The notion of identity is relevant even in such a "single assignment" language, although it is much more important when objects are modified.

Strong integrity is more difficult to define concisely in positive terms. One typical breach of object integrity is that typing is not strong enough: e.g. in C++ it is easy to handle areas of storage without any regard to the type of objects stored there. Another kind of breach is that superclass subobjects may be arbitrarily split in multiple inheritance, e.g. in Flavors and Eiffel (Ch. 4 §3.8).

The deletion of objects raises an important question at once: should it be requested explicitly, or happen automatically when objects become unreachable? It is sometimes claimed that existence by reachability and garbage collection (or some other kind of automatic storage management) are essential features of OOPLs. For instance, Zdonik and Maier's reference model for OODBs [Zdonik &c 90 p. 13] has persistence by reachability (from the root object of the database) as one requirement. I disagree nevertheless with this view (§4).

2. An overview of the constituent papers — with hindsight

2.1. General

Of the five papers in this dissertation, the fourth and fifth (Chapters 5 and 6) are very recent. The three other articles are already so old (3 □ 4 years) that I have been able to rethink about their topics. The corresponding subsections will therefore be longer.

Looking at the titles of the papers (chapters), a reader might legitimately think that a more appropriate title for the collection could be "My combat against C++", and that the thesis lacks more general interest. The attitude is not adamantly negative, however: e.g., the inheritance features of C++ are judged mostly superior to those of competing languages in Ch. 4. I also think that there are at least two good reasons why this thesis can be relevant to other people besides C++ insiders.

First, the discussion in the articles is mostly based on more general principles and comparisons with many other languages. Many of the important conclusions and suggestions, especially in Ch. 5, are applicable much more widely than only to C++. I have tried to avoid treating details that are marginal to the principles of good programming, although they might be very important for some specific applications. A good example is the current hot debate on whether and how an exponentiation operator should be added to C++ — without doubt an important feature for numeric programming.

Second, as the foundations of object-oriented programming have not been definitely laid out yet, learning from the experiences (especially negative experiences) of existing languages is one good way to progress. Pointing out defects in the existing languages should also help against complacency and make people think that progress *is* still needed. Today there seems to be a real danger of premature fixation and standardisation on C++ in the OOP community; the situation has changed dramatically from the time when Ch. 2 was written.

All constituent papers are very informal. I do not think that formalism is always necessary as a goal in itself. Evidently, some of the ideas in Ch. 4 and 5 could be fruitful starting points for more rigorous treatment in the future, but that job is better suited for researchers who are already well versed in appropriate formal methods and tools. I felt sympathy with the following note [Madsen 92a]:

> During the design of BETA we were often criticized for not developing a formal semantics together with the language. The reason we were not doing this was that formal semantics did not have anything new to offer. The main contribution of formal semantics is a clarification of well known language constructs. When new language constructs are invented it is very useful to refine them using formal theories.

2.2. On the darker side of C++

As briefly mentioned in §1 and §13 of Ch. 2, the first paper arose from my experiences in trying to implement parts of Airi Salminen's document database model [Salminen 87, 89] in C++. It gradually dawned upon me that this work was near something that was called "object-oriented programming". C++ was not generally regarded as a mainstream OOPL in 1987 □ 1988, at least not in Europe. It was not possible to compare C++ extensively to other more or less object-oriented languages at that point. I had no experience with any of those languages, and there was not so much literature available. For instance, the landmark book [Meyer 88] had not appeared yet.

This chapter thus analyses C++ mainly with two points of reference: established conventional languages such as Pascal and Ada, and a budding general philosophy of object orientation. Perhaps the most important point of the whole article is that C is an unsound basis for object-oriented extensions. A large number of defects and problems, of varying degrees of severity, in C++ as it was at the time (before Release 2.0), are described. In most cases, I give either a suggestion for improving the language, some advice to programmers to work around the problem, or both.

A few of the flaws noted in Ch. 2 were obviously incurable. For some others again, corrections had already been advertised to come in the next release. Those features that fall in between these two extremes are interesting concerning the possible effects of my article. Indeed, a few things have been improved more or less as I had suggested: e.g., the different integral types are now clearly distinct (Ch. 2 §3), and member functions are not allowed to modify constant objects (Ch. 2 §11). These ideas are so obvious that they can have occurred even earlier to the developers of C++, of course.

I have not noted anything essential in Chapter 2 behind which I cannot still stand today. There are some small slips in details: e.g. 'restricted private types' of Ada in Ch. 2 §2 instead of 'limited private types'. The constructor in the 'flexstring' class presented as an example in Ch. 2 §8 is a little too streamlined: one should first check whether **this** is non-null, implying that the object is not being created on the heap and therefore its size is already fixed. However, the whole original method of dynamic storage allocation and deallocation has been superseded by much better principles in newer releases of C++.

2.3. Comments on "the Law of Demeter" and C++

The second paper (Ch. 3) treats a question that can concern either object-oriented programming languages themselves, their programming environments, or just the style and discipline to which programmers may adhere. This paper was important for me because it began a co-operation with the Demeter group at Northeastern University, which has continued with varying intensity ever since.

The article tries to clarify and criticise the Law of Demeter as originally presented. The critique about not being always able to verify conformance to the law may be unnecessarily strong. The main contribution of Ch. 3 is, as the title suggests, in examining some alternative ways for interpreting the law in a hybrid language, especially C++. Also, the relationship of the law to untyped languages is discussed, as it had been originally formulated for typed languages. (I have been later persuaded by some people to speak of *statically typed* vs. *dynamically typed* languages rather than typed vs. untyped.)

The idea of *acquaintances* is proposed in Ch. 3 §3 and §7 as a direct analogue of *friends* in C++. While an outside class or function needs to be a friend of a class A in order to have access to A's private components, it would need to be an acquaintance of A in order to access its public components. This would happen by declaring A as a *known* class in the header of the other class or function.

It must be confessed that the reasoning is rather opaque and hard to follow in some parts, e.g. Ch. 3 §4 and §8. However, the conclusions and suggestions still appear sensible to myself.

The critique and ideas of Ch. 3 were taken into account in [Lieberherr &c 89], which developed further the Law of Demeter. The idea of acquaintances from Ch. 3 was accepted, but note that meaning of the term was inverted to conform to its usage in Hewitt's actor model: the acquaintances of a class are those other classes that it is allowed to access.

The main restriction set by the Law of Demeter is that one should not see more than one level of the structure of a complex object from one viewpoint. There may be one important situation in which this requirement is conceptually invalid; it has not been recognised as an exception in Ch. 3, nor in [Lieberherr &c 89]. Namely, several consecutive levels of structure may sometimes model the structure of objects in the real world, and should not be hidden from the programmers. Such part hierarchies had been discussed already in [Blake &c 87]; one example from that paper is

```
joe leftLeg.foot.bigToe.wiggle
```

The hindsight here is that I had not seen [Blake &c 87] when writing Chapter 3. Such situations can be handled nicely in the current Demeter system by propagating interfaces through the class hierarchy [Lieberherr 92].

This last observation is directly analogous to the fact that the *weak* variant of the Law is more appropriate to some cases than the *strong* variant, which forbids direct access to inherited instance variables. This aspect is treated thoroughly in Ch. 4.

2.4. Disciplined inheritance

The working title of the third paper (Ch. 4) was "Inheritance considered harmful", when I began to write it. It appeared at ECOOP'89 that some other authors had thought about the same name for their contributions, but all had been sensible enough to change it.

The main objectives of Ch. 4 are to subordinate inheritance to the more important principles of object orientation, to classify it into at least two clearly different variants, and to warn against its overuse. As a reaction to the common overemphasis on inheritance, I try to "explain it away" as far as possible, by aggregation and some constraints.

It should be noted that late binding is not counted as an inherent facet of inheritance, but as a separate feature. This conforms to [Atkinson &c 89], while [Cook 89] takes the opposite view: the essence of inheritance. I will treat this more in the next section.

The two opposite cases, incidental and essential inheritance, are recognised in the article. The former corresponds approximately to *implementation inheritance*, the latter to *interface inheritance*; these are terms employed by many authors. Incidental inheritance is shown to be equivalent to aggregation with three simple constraints; the difference between a component (instance variable) and a superclass part (subobject) is thus very small.

In hindsight, it is an exceptionally strong requirement in Ch. 4 §6 that late binding be suppressed in incidental inheritance. I probably had the misconception (and probably not I alone) that this were the case in the *private* inheritance of C++. When writing Chapter 5, it became clear to me that C++, on the contrary, allows even *too far-reaching* late binding of virtual functions in private inheritance. — It is consistent with the strong requirement to require also that the *protected* operations of the superclass not be invokable from the subclass in incidental inheritance (Ch. 4 §6).

The three constraints listed in Ch. 4 §5 are not tied together by logical necessity, as noted in Ch. 4 §9. The relationship between components and parents (aggregation and inheritance) has meanwhile been extensively studied in [Taivalsaari 91]. Today, I would relax the above requirement and distinguish between incidental inheritance and mere aggregation on the basis of whether late binding is possible or not. Consequently, there is no difference when the superclass has no virtual operations!

I maintain now, even more strongly than expressed in the paper, the opinion that *essential* inheritance is the more important and interesting kind. This has often been regarded as a typical European attitude, while many American researchers favour the viewpoint of implementation reuse. Essential inheritance, implying an *is-a* or at least *is-like* [Wegner &c 88] relationship, does not rely on late binding for its definition. Indeed, it is the kind of inheritance that is used in knowledge representation, where operations in the OOP sense do not normally exist [Touretzky 86].

I also continue to consider it desirable that all operations need not be virtual (although virtuality could be the default, contrarily to the practice of Simula and C++), and even that late binding can be suppressed in invocations of virtual operations within the same class. There is a somewhat opposite opinion in [Cook 89 §4.4]:

> Local access, which performs the effect of a *z* operation but does not use the virtual *z* component, should be used only when the operation is not properly viewed as an abstract use of the *z* operation. It is almost impossible to justify using an operation in any way but its abstract form.

As long as non-virtual operations are allowed, the question about the temporary early binding of virtual operations could even be considered moot. Namely, a class designer can decompose each virtual operation 'oper' so that there is a private non-virtual counterpart 'oper_static' and 'oper' only calls 'oper_static' with its own arguments. Instead of an early-bound call of 'oper', one just calls 'oper_static' directly. — In strict inheritance (§3.2), there can never be a need to force early binding.

The main reason why I like the possibility of early binding is a "humble programmer" [Dijkstra 72] viewpoint. There is not even a general definition of when a redefinition of an operation in a subclass behaves similarly enough to the superclass operation. The assertion redefinition rule [Meyer 88 §11.1.2] helps *if* the pre- and postconditions have been specified well enough, but assertions cannot in general be checked statically. Most OOPLs do not even support assertions or any similar mechanism. A class designer can therefore have more confidence in getting the expected behaviour of an operation with early than late binding.

We could say that here lies a *fundamental dilemma of virtual operations* (in non-strict inheritance): should the operation behave consistently mainly with its class of definition (i.e., its lexical environment) or with the (sub)class on whose instance it is invoked (its dynamic environment)? The former goal favours static binding of other operations invoked on **self** (**this**), the latter goal, dynamic binding.

The analysis of *fork-join inheritance* (Ch. 4 §8), although short, might be the most important contribution of the paper. It is shown that the typical published and implemented approaches to multiple inheritance, including that of Eiffel, violate the integrity of superclass subobjects. The multiple inheritance principles that had been announced for C++ are completely sound in this respect. Finding faults even in those principles became later the main theme of the following paper included in this dissertation.

2.5. A critique of the inheritance principles of C++

The initial idea for the fourth paper (Ch. 5) was born during my one-week visit at Northeastern University in the autumn of 1990, inspired by discussions with several people there. The original rule was:

> Public inheritance should be "virtual" (shared), private inheritance "non-virtual" (duplicated).

Of course, this was a little too crude because late binding applies even to private inheritance (§2.4). An essential distinction is therefore made in the chapter between *accessible* and *inaccessible* superclasses (Ch. 5 §3.3). All immediate superclasses of a class C are accessible to C itself, but private superclasses are inaccessible to subclasses of C. The final refinement is presented as the Rule in Ch. 5 §5.4 — the main result of the paper.

The problem with private inheritance mentioned in the previous subsection is solved with the requirement (Thesis 1) that a descendant class must not be able to redefine a virtual function of an inaccessible ancestor class (Ch. 5 §3.4). This is one of the few suggestions that concern already single inheritance. Most others purport to improve the semantics of multiple inheritance.

An anomalous consequence of the current C++ rules, which was truly surprising to myself at first, is presented in Ch. 5 §5.6 and called "the exponential yoyo problem". An interesting although remote analogy are some of the anomalies shown in [Baker 91] about the linear superclass precedence ordering of CLOS, e.g.

> A generic function may require O(n!) method combinations for a class having only O(n) methods defined.

The article is constructive in the sense that it offers a concrete suggestion of remedy to each claimed defect or problem. My guess is that the proposed modifications would not have detrimental cross effects to other parts of C++, nor cause a large proportion of existing C++ software to change its behaviour or become illegal. — Many of the suggestions could actually be realised by mere programming discipline. Some others, notably the main rule, absolutely require changes to the language itself.

About the initialisation of shared (virtual) superclass subobject, Ch. 5 §3.6 just says that the current C++ rule is inconvenient. It has already appeared that it is not so simple to invent a better rule as I had supposed [Sakkinen 92b].

Since I had long believed in the utility of multiple inheritance, the results of Ch. 5 were encouraging. They are also rather obviously generalisable to other object-oriented languages. Doing that and making the treatment more rigorous is one of my next research goals.

2.6. The darker side of C++ revisited

The fifth paper (Ch. 6) tries to take a broad look at disadvantages not covered by Ch. 5 and mostly not treated in Ch. 2 either. It can thus not be highly focussed. It is the most political and most opinionated of the constituent papers, but I tried to give technical backing to the political statements. A reason for the partisan attitude was given in §2.1.

The prime argument of Ch. 5 is that C++ is insufficiently object oriented because its objects do not contain enough run-time type information. The consequences are unsafety, type loss, and impossibility of unconstrained polymorphism. This disadvantage was only quite briefly mentioned in Ch. 2. I continue to point out the compatibility with C as the other serious (if not fatal) defect of C++. In particular, new harmful aspects of pointer handling in C and C++ are pointed out.

One well-known further problem of pointer arithmetic, or of its harmful interference with the object-oriented features of C++, that would have been worth mentioning in Ch. 6 §2.4 is the following: If T is a *class* type and S a subclass of T, a pointer of type T* can legally point also to an instance of S. However, no address arithmetic is allowed in that case even if the referent object is an array element. The reason is that instances of S are usually larger than instances of T.

Since the C community has a long-standing habit of regarding Ada as an overly complex monster, I did not omit the opportunity to remark that C++ has already grown to the same order of complexity — even though it does not yet tackle the problems of concurrency!

3. More about inheritance

3.1. Class inheritance

Class-based languages are even today the mainstream of object-oriented programming, and all constituent papers of this dissertation concentrate on them. It is therefore natural to restrict this section to the class approach as well. Inheritance is then a relationship between classes, not instances.

The identity concept was stressed as fundamental in §1, for instances. The essence of identity is that two objects can have exactly the same value or contents and still be distinguishable. Consequently, if classes are also objects (as in Smalltalk-80), two classes can be completely alike but still distinct from each other. Even in those languages in which classes are not first-rate objects, *structural class (type) equivalence* can be seen as contrary to object orientation. There is an enjoyable discussion of its good and bad sides in [Nelson &c 91 §8.1]; Modula-3 does use structural equivalence, but object orientation is indeed not its main goal.

An example originally designed to show the advantages of static typing [Magnusson 91] happens to illustrate my point:

```
CLASS Circle
    operations: Draw, Move
CLASS Cowboy
    operations: Draw, Move, Shoot
```

With structural equivalence, Cowboy without the Shoot operation would be equal to Circle, assuming that even the signatures of the operations conform. The additional operation makes it a subtype (subclass) of Circle. This would not hold in most statically typed object-oriented languages. Structural equivalence thus loses some semantic power.

Therefore, I support "name equivalence" or rather *declaration equivalence* for classes. That does not exclude the possibility of later identifying two classes that have been declared separately but are exactly alike, whether they have the same name or not. (This is fully analogous to the coalescing of instances [Khoshafian &c 86].) A language could let the programmer decide for each class separately whether structural or declaration equivalence should be applied to it .

In the same vein, the term 'inheritance' should be used according to the prevailing tradition, i.e. only when a subclass is explicitly declared to inherit a superclass, or perhaps vice versa [Pedersen 89]. There are some approaches in which two classes may be said to have an inheritance relationship although they have been defined fully independently of each other. Such relationships are better called 'subtyping' or 'conformance'.

There are also approaches in which subtyping and inheritance are completely separated. This brings some advantages: very often the implementation hierarchy tends to be opposite to the subtype hierarchy [Snyder 87]. Unfortunately, subtyping usually seems to be based on structural type equivalence [Cook &c 90; America &c 90]. Semantic problems of the same kind as in the above example then remain.

3.2. Strict inheritance

Inheritance is called *strict* if descendants do not delete or modify (override) any inherited features, *non-strict* otherwise. While people are probably aware that a large part of the problems connected with inheritance are caused by non-strictness, we mostly seem to think that strict inheritance is much too limited to be interesting. Here I try to indicate briefly some ways in which strict inheritance could be beneficial. Essential (public) inheritance is assumed on the first stage.

There may be useful cases of *independent* multiple inheritance in which the subclass is simply a Cartesian product of its immediate superclasses, i.e. it does not even need to add any instance variables or operations. Any name clashes between features inherited from different superclasses should be regarded as casual [Knudsen 88], i.e. simple overloadings. An often-discussed example would be the class Coloured_point that inherits Coloured_object and Point (cf. §3.4). — In *fork-join* inheritance, there shall be only one subobject of each non-immediate ancestor class. A typical simple case consists of the classes Person, Student, Employee, and Student_employee (Ch. 5 §5.2).

With strict inheritance, the devious problems of inheritance graph linearisation [Snyder 87; Baker 91] simply disappear! When casual name clashes are disambiguated, there is always only one superclass in which a given feature is defined. Linearisation could be made consistent even in non-strict inheritance by the following constraint: if two classes A and B both redefine some operation inherited from a common ancestor, and neither of them is a subclass of the other, then no class must inherit both A and B. This rule would make '*super*' (in the Smalltalk sense) unambiguous even with multiple inheritance.

I imagine that many typical class hierarchies, which contain a lot of operation redefinitions in subclasses, could be reorganised so that all inheritance becomes strict. Generalisation mechanisms [Pedersen 89] could aid in such reorganisation. Pedersen's proposal has the disadvantage that every new class created by generalisation becomes a top-level class, i.e., has no superclasses. This disadvantage is negligible in strict inheritance.

It is also possible to design classes originally in such way that later subclasses only rarely need to redefine inherited operations. A typical class such as Person can actually consist of a set of *kernel* features and another set of *default* features. The former are those that the class designer would, even with afterthought, claim essential also to all imaginable subclasses of Person. The latter are those that allow instances of Person to be created before any subclasses are defined; this is necessary for typical object-oriented software development. — Now, simply keep only the kernel features in class Person, and define a subclass Default_person, which contains the default features. It might even appear useful to relegate the least universal features further into Standard_person, which would be a subclass of Default_person. Alternatively, the defaults could be divided into several parallel subclasses.

This reasoning is heading in the direction of *fine-grain inheritance* [Johnson &c 92], which aims to a large number of very simple classes with a "dense" multiple-inheritance graph. It seems that it would be relatively easy to make fine-grain inheritance strict.

Finally, let us look at incidental (private) inheritance. It is utterly non-strict toward clients, since none of the superclass operations can be invoked on a subclass object unless explicitly re-exported. However, it is sensible to call an incidental inheritance relationship *inward strict* if the subclass does not redefine any superclass operations nor make them unavailable *inside* a subclass object. The advantages of strictness actually come from the non-redefinability.

3.3. Deferred (abstract) classes

The most common term for a class that cannot have direct instances is 'abstract class', and its opposite thus 'concrete class'. Unfortunately the word 'abstract' has so many mutually related meanings that I will prefer 'deferred class' and its opposite 'effective class' [Meyer 88 §10.3].

Interestingly, the Demeter system [Lieberherr &c 91] requires an even stronger division of conventional classes than the kernel vs. default split recommended in the previous subsection: it is not allowed to inherit effective classes at all. On the other hand, Demeter does not require strict inheritance.

There are different degrees of deferredness. The ultimate is a mere *interface*: operation signatures and possibly some assertions. Another important kind are classes which can additionally contain implementations of operations but no instance variables. These are called 'protocols' in [Whitewater 91]: yet another overloading of that term. I will call these, 'pure deferred classes'. Of course, a class can in general define (or inherit) even instance variables and still be deferred. The approach of [Dodani&c 92] forbids this, however: all deferred classes must be pure deferred.

Essential (public) inheritance from deferred classes is largely the opposite of incidental (private) inheritance from effective classes. In the former case, parts of the functionality of the superclass are implemented by subclasses. In the latter case, parts of the functionality of the subclass are implemented by superclasses. The next subsection will try to look at inheritance from an even more symmetric viewpoint.

Regarding interface classes as *types* looks to me to be a more object-oriented alternative to the complete separation of inheritance and subtyping (§3.1). To be precise, an interface class could be regarded as a type if and only if no other classes that are not themselves types appear in its signature. — Interfaces are behaviourally underspecified [Wegner &c 88]. One could be more general and regard all pure deferred classes as types; they could then be overspecified.

A common problem in statically typed languages is that the creation of a new object must be statically bound (to an effective class), while one would often like to have late binding just as in calling virtual operations [Koskimies &c 91]. In the most usual case in which the actual class of the new object should be the same as that of an existing object, the task is easy in Smalltalk and similar languages:

someObject **class new**

The way to achieve this in e.g. C++ is obvious but hardly "easy" as advertised in [Stroustrup 91 §6.7.1]: one must remember to redefine a virtual function that creates the new object, in every subclass.

There are no deep reasons why the actual class of an object to be created could not be late-bound even in a statically typed language. Indeed, a mechanism is presented in [Palsberg &c 91a] which automatically binds directly recursive occurrences of **new** to the dynamic class of the invoking object.

3.4. Mix-in inheritance

Mix-in classes are widely used in LISP-based object-oriented languages. A mix-in is characterised in [Bracha &c 90] as an *abstract subclass*, similarly to abstract (deferred) superclasses. Terms such as 'difference class' [Stein 88] have also been used. The main problem with mix-ins seems to be that they complicate method lookup and method combination.

The situation becomes simple if we restrict ourselves to *strict* mix-in inheritance (§3.2). We then get a true "class algebra" [Stein 88] (see Ch. 4 §5): since redefinitions are forbidden, no distinction between subclass and superclass is needed. An effective class may be defined as the Cartesian product of a set of (possibly deferred) component classes C_1, ..., C_n provided that:

- No operation is defined as public (exported) in more than one class.
- Every operation that is deferred (imported) in some class C_i is defined (exported) in some other class C_k.

This becomes more useful but also a little more complicated if we require only *inward* strictness. The visibility of every public operation of every component class can then be specified, to other component classes and to clients of the product class. Even multiple definitions of operations can be allowed if their scopes do not overlap. Here we have again an idea that would seem to apply well to fine-grain inheritance (§3.2).

There is a problem presented in [Stein&c 89 §3.5], which is said to lack a fully satisfactory solution in existing class-based languages. Let there be an arbitrary hierarchy of subclasses of Shape, representing different kinds of geometric objects, and suppose that an amount of software using that hierarchy has been built. At that point, we would like to add a colour attribute to new geometric objects, without having to modify the existing classes. Defining a separate Colour_mixin class is better than directly defining a subclass Coloured_shape, but in any case coloured versions of all classes in the hierarchy must be produced.

A possible way to handle a class hierarchy as a unit and make a "parallel" hierarchy by one definition is briefly sketched in [Cook 89 §3.5]. The idea looks far from fully developed, however. Another solution would be to allow an object to be a direct instance of more than one class [Sakkinen 90b §3]. The Colour_mixin class could then be used also together with classes that are not descendants of Shape.

3.5. Some related research

The dissertation [Cook 89] examines and explains inheritance using formal semantics, as is evident from the title. The redefinition and late binding of operations is taken as the most important facet of inheritance, unlike I am doing here. Actually, late binding is mostly regarded as concerning the self-reference of objects, as in *delegation*. A nice concise definition from the chosen viewpoint is given in [Cook 89 §1.1]:

> Inheritance is a mechanism for incremental programming in the presence of self-reference.

It is tacitly assumed here that the self-reference is late-bound.

Cook treats many interesting topics. For instance, he shows [Cook 89 Ch. 9] that the relationship between subclasses and superclasses in BETA is almost the inverse of that in most other OOPLs. It is also pointed out that operation redefinition in BETA is much more disciplined than in the mainstream languages, because it is controlled by '**inner**' in the superclass and no complete replacement is possible. (However, 'inner' is a genuine imperative: it may be executed many times in the same superclass operation.)

The thesis [Taivalsaari 91] takes a very different approach than Cook. It is much more informal, but also more constructive, e.g. presenting three possible new models for object-oriented languages [Taivalsaari 91 Ch. 5]. The interest is more on mechanisms than conceptual modelling, as the title implies. Inheritance is examined on two levels: interface inheritance and property inheritance [Taivalsaari 91 Ch. 4]. Additionally, the distinction between class-based and prototype-based languages is made orthogonal to the distinction between delegation and *concatenation*.

The disadvantages of class hierarchy linearisation in multiple inheritance were well explained already in [Snyder 87]. One approach for avoiding those disadvantages without always needing explicit disambiguations of multiply defined names is presented in [Carré &c 90]. I had not happened to read that paper before writing Chapter 5. It seems that the "point of view" method could handle situations with possible "sideways inheritance" (Ch. 5 §5.5) more flexibly. On the other hand, it would evidently not solve the problem that mutually unrelated operations with the same name (and signature) are forcibly unified (Ch. 5 §4.1).

4. Other topics of further research

The *ontology* of objects is an interesting area. Stressing object identity as one of the prime concepts is already a stand on the question what it means for an object to exist. As mentioned in §1.4, I do not believe that the garbage collection of unreachable objects is the only correct way to remove unnecessary objects from a system. Rather the opposite: in *value-oriented* systems, objects can be regarded as auxiliary entities whose existence or deletion is merely a pragmatic issue; in an object-oriented system, both the creation and the destruction of an object can be semantically meaningful events.

Many object-oriented languages offer no means for the direct deletion of objects; a serious conceptual defect in my opinion. The significant pragmatic advantage gained is of course that dangling references need not be worried about. I began to study possibilities to reconcile the principles of garbage collection and explicit (possibly cascading) deletion in [Sakkinen 88b], by "positive" and "negative" existential dependences between objects. I am planning to continue that work in the near future.

Ensuring the *locality*, or preventing too wide propagation, of object operations is a problem that is not very often mentioned. This is the additional complexity that pure abstract datatypes have not got (§1.2). For instance, the Law of Demeter only restricts the direct visibility of objects to operations, not how far their effects may reach indirectly. An interesting recent paper in this area is [Hogg 91]. The problem looks really fundamental: I do not believe that any complete and general solution exists.

Related to the locality problem is the problem of indirect recursion in the following meaning: Suppose that some operation O of an object A calls an operation P of another object B. Object A cannot know whether that call causes one or more other operations (another O not excluded) of A to be invoked before control returns to the point after the call of P. If we have assertions in the style of Eiffel [Meyer 88 §7.4; Meyer 92 Ch. 9], there remains a surprising gap: an object invoking some operation on another object cannot in general be sure that even the *class invariant* of the callee holds, because an "outer" operation on it may be in progress somewhere on the call stack. — Instance-wise direct recursion, i.e. calls on **this** (**self**), is most typical object-oriented programming; it is not similarly devious as the indirect case.

Combining classes and prototypes is a research pursuit that concerns dynamic object-oriented languages, and not the statically typed, compilable variety that I have mostly studied. However, the idea of "titles", which was originally invented to disambiguate name conflicts, rather naturally lead to such extensions [Sakkinen 90b]. This may not be one of the most urgent topics for me to develop further.

Enumerated types are not common in object-oriented languages; however, at least C++ and Modula-3 support them. In [Sakkinen 91b] I argued strongly in their favour as a simple abstraction mechanism complementary to classes. The difficulties of extending enumerations have been one major cause why they have been abandoned in some recent languages. I now have some initial ideas about extensible enumerations. An overlapping problem is the relationship of enumerations to inheritance.

For a long time already, I have tried to generalise the concept of *order* between values and between objects [Sakkinen 87, 88a, 90a]. I still think that more general orderings than linear and partial order within collections can be useful, especially in object-oriented database systems. For instance, the just-mentioned problem of extending enumerated types can be solved more consistently if they are either completely unordered or quasi-ordered (preordered).

I mentioned the fascination of object orientation already in §1.1. For a researcher, almost every solved problem seems to raise equally interesting new questions and possibilities. For a language designer and a practitioner, the fascination may be even dangerous. My current view is that the features and facilities offered by object-oriented languages must be used with intelligence and in moderation for object-oriented programming to really result in better software, more easily created and maintained than with the tools and methods of the past.

References

[ACM 87] *ACM Turing Award Lectures: The First Twenty Years: 1966 □ 1985*. ACM Press 1987.

[Allen 89] F.E. Allen (Chair). *POPL '90 (Principles of Programming Languages) Proceedings*. ACM Press 1990.

[America 91] Pierre America (Ed.). *ECOOP '91 (European Conference on Object Oriented Programming) Proceedings*. Springer-Verlag 1991 (LNCS 512).

[America &c 90] Pierre America, Frank van der Linden. "A Parallel Object-Oriented Language with Inheritance and Subtyping". [Meyrowitz 90], 161 □ 168.

[Atkinson 91] Colin Atkinson. *Object-Oriented Reuse, Concurrency and Distribution : An Ada-based Approach*. ACM Press 1991.

[Atkinson &c 89] Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, Stanley Zdonik. "The Object-Oriented Database System Manifesto". [Kim &c 89b], 40 □ 57.

[Baker 91] Henry G. Baker. "CLOStrophobia: Its Etiology and Treatment". *ACM OOPS Messenger* Vol. 2 No. 4 (October 1991), 4 □ 15.

[Bézivin &c 87] Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, Henry Lieberman (Eds.). *ECOOP '87 (European Conference on Object Oriented Programming) Proceedings*. Springer-Verlag 1987 (LNCS 276).

[Blake &c 87] Edwin Blake, Steve Cook. "On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk". [Bézivin &c 87], 41 □ 50.

[Bracha &c 90] Gilad Bracha, William Cook. "Mixin-based Inheritance". [Meyrowitz 90], 303 □ 311.

[Carré &c 90] Bernard Carré, Jean-Marc Geib. "The Point of View notion for Multiple Inheritance". [Meyrowitz 90], 312 □ 321.

[Cook 89] William R. Cook. *A Denotational Semantics of Inheritance*. Dissertation, Brown University (Providence, RI, USA) 1989.

[Cook &c 90] William R. Cook, Walter L. Hill, Peter S. Canning. "Inheritance Is Not Subtyping". [Allen 89], 125 □ 135.

[Danforth &c 88] Scott Danforth, Chris Tomlinson. "Type Theories and Object-Oriented Programming". *ACM Computing Surveys* Vol. 20 No. 1 (March 1988), 29 □ 72.

[Dijkstra 72] Edsger W. Dijkstra. "The Humble Programmer". *CACM* Vol. 15 No. 10 (October 1972), 859 □ 866. Reprinted in [ACM 87], 18 □ 31.

[Dodani &c 92] Mahesh Dodani, Chung-Shin Tsai. "ACTS: A Type System for Object-Oriented Programming Based on Abstract and Concrete Classes". [Madsen 92b], 309 □ 328.

[Fetzer 88] James H. Fetzer. “Program verification: the very idea”. *CACM* Vol. 31 No. 9 (September 1988), 1048 □ 1063.

[Gjessing &c 88] S. Gjessing, K. Nygaard (Eds.). *ECOOP ’88 (European Conference on Object Oriented Programming) Proceedings*. Springer-Verlag 1988 (LNCS 322).

[Hogg 91] John Hogg. “Islands: Aliasing Protection In Object-Oriented Languages”. [Paepcke 91], 271 □ 285.

[Johnson &c 92] Paul Johnson, Ceri Rees. “Reusability Through Fine Grain Inheritance”. Technical report, GEC-Marconi Research (Chelmsford, England) 1992.

[Kangassalo &c 90] Hannu Kangassalo, Setsuo Ohsuga, Hannu Jaakkola (Eds.). *Information Modelling and Knowledge Bases*. IOS Press 1990.

[Khoshafian &c 86] Setrag N. Khoshafian, George P. Copeland. “Object Identity”. [Meyrowitz 86], 406 □ 416.

[Kim &c 89a] Won Kim, Frederick H. Lochovsky (Eds.). *Object-Oriented Concepts, Databases, and Applications*. ACM Press 1989.

[Kim &c 89b] Won Kim, Jean-Marie Nicolas, Shojiro Nishio (Eds.). *DOOD ’89 (Deductive and Object-Oriented Databases) Proceedings*. Elsevier 1989.

[King 89] Roger King. “My Cat is Object-Oriented”. [Kim&c 89a], 23 □ 30.

[Knudsen 88] Jørgen Lindskov Knudsen. “Name Collision in Multiple Classification Hierarchies”. [Gjessing &c 88], 93 □ 109.

[Knudsen &c 88] Jørgen Lindskov Knudsen, Ole Lehrmann Madsen. “Teaching Object-Oriented Programming is More than Teaching Object-Oriented Programming Languages”. [Gjessing &c 88], 21 □ 40.

[Koskimies &c 92] Kai Koskimies, Juha Vihavainen. “The Problem of Unexpected Subclasses”. *Journal of Object-Oriented Programming*, to appear.

[Kristensen &c 87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, Kristen Nygaard. “The BETA Programming Language”. [Shriver &c 87], 7 □ 48.

[Lieberherr &c 89] Karl Lieberherr, Ian Holland. “Formulations and Benefits of the Law of Demeter”. *ACM SIGPLAN Notices* Vol. 24 No. 3 (March 1989), 67 □ 78.

[Lieberherr &c 91] Karl J. Lieberherr, Paul Bergstein, Ignacio Silva-Lepe. “From objects to classes: algorithms for optimal object-oriented design”. *Software Engineering Journal* Vol. 6 No. 4 (July 1991), 205 □ 228.

[Lieberherr 92] Karl Lieberherr. Private communication, 1992.

[Liskov 88] Barbara Liskov. “Data Abstraction and Hierarchy”. [Power &c 88], 17 □ 34.

[MacLennan 82] B.J. MacLennan. "Values and objects in programming languages". *ACM SIGPLAN Notices* Vol. 17 No. 12 (December 1982), 70□79. Reprinted in [Peterson 88], 9 □14.

[Madsen 92a] Ole Lehrmann Madsen. Unpublished workshop contribution, Århus (Denmark) 1992.

[Madsen 92b] Ole Lehrmann Madsen (Ed.). *ECOOP '92 (European Conference on Object Oriented Programming) Proceedings*. Springer-Verlag 1992 (LNCS 615).

[Magnusson 91] Boris Magnusson. "Adjusting the Type-Knob". [Palsberg &c 91b], 44 □ 48.

[Masini &c 91] Gérald Masini, Amedeo Napoli, Dominique Colnet, Daniel Léonard, Karl Tombre. *Object-Oriented Languages*. Academic Press 1991 (A.P.I.C. Series, No. 34).

[Meyer 88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall 1988.

[Meyer 92] Bertrand Meyer. *Eiffel: the Language*. Prentice Hall 1992.

[Meyrowitz 86] Norman Meyrowitz (Ed.), *OOPSLA (Object-Oriented Programming: Systems, Languages, and Applications) '86 Proceedings. ACM SIGPLAN Notices* Vol. 21 No. 11 (November 1986).

[Meyrowitz 87] Norman Meyrowitz (Ed.), *OOPSLA '87 Proceedings. ACM SIGPLAN Notices* Vol. 22 No. 12 (December 1987).

[Meyrowitz 88] Norman Meyrowitz (Ed.), *OOPSLA '88 Proceedings. ACM SIGPLAN Notices* Vol. 23 No. 11 (November 1988).

[Meyrowitz 89] Norman Meyrowitz (Ed.), *OOPSLA '89 Proceedings. ACM SIGPLAN Notices* Vol. 24 No. 10 (October 1989).

[Meyrowitz 90] Norman Meyrowitz (Ed.). *OOPSLA/ECOOP '90 Proceedings. ACM SIGPLAN Notices* Vol. 25 No. 10 (October 1990).

[Nelson &c 91] Greg Nelson (Ed.). *Systems Programming in Modula-3*. Prentice Hall 1991.

[Paepcke 91] Andreas Paepcke (Ed.). *OOPSLA '91 Proceedings. ACM SIGPLAN Notices* Vol. 26 No. 11 (November 1991).

[Palsberg &c 91a] Jens Palsberg, Michael I. Schwartzbach. "What is Type-Safe Code Reuse?". [America 91], 325 □ 341.

[Palsberg &c 91b] Jens Palsberg, Michael I. Schwartzbach (Eds.). *Types, Inheritance and Assignments* (ECOOP'91 workshop papers). Aarhus University (Denmark) 1991.

[Pedersen 89] Claus H. Pedersen. "Extending Ordinary Inheritance Schemes to Include Generalisation". [Meyrowitz 89], 407 □ 417.

[Peterson 88] Gerald E. Peterson (Ed.). *Tutorial: Object-Oriented Computing, Volume 1: Concepts*. IEEE Computer Society Press 1988.

[Power &c 88] Leigh Power, Zvi Weiss (Eds.). *OOPSLA '87 Addendum to the Proceedings. ACM SIGPLAN Notices* Vol. 23 No. 5 (May 1988).

[Sakkinen 87] Markku Sakkinen. "Comparison as a Value-yielding Operation". *ACM SIGPLAN Notices* Vol. 22 No. 8 (August 1987), 105 ◘ 110.

[Sakkinen 88a] Markku Sakkinen. "Generalised order concepts for databases". Manuscript, University of Jyväskylä 1988.

[Sakkinen 88b] Markku Sakkinen. "Objects, non-objects, and existential dependences". Manuscript, University of Jyväskylä 1988.

[Sakkinen 90a] Markku Sakkinen. "Modelling order in databases". [Kangassalo &c 90], 177 ◘ 202.

[Sakkinen 90b] Markku Sakkinen. "Between classes and instances, aided by titles". Manuscript, University of Jyväskylä 1990.

[Sakkinen 91a] Markku Sakkinen. "On treating Basic and Constructed Types Uniformly in OOP". [Palsberg &c 91b], 60 ◘ 63.

[Sakkinen 91b] Markku Sakkinen. "Another defence of enumerated types". *ACM SIGPLAN Notices* Vol. 26 No. 8 (August 1991), 37 ◘ 41.

[Sakkinen 92a] Markku Sakkinen. *Multiple Inheritance and Multiple Subtyping* (ECOOP'92 workshop papers). University of Jyväskylä 1992.

[Sakkinen 92b] Markku Sakkinen. "On the initialisation of shared superclass parts". [Sakkinen 92a], 42 ◘ 43.

[Salminen 87] Airi Salminen. "A Relational Model for Unstructured Documents". [Yu &c 87], 196 ◘ 207.

[Salminen 89] Airi Salminen. *A Model for Document Databases*. Dissertation, University of Jyväskylä 1989.

[Shriver &c 87] Bruce Shriver, Peter Wegner (Eds.). *Research Directions in Object-Oriented Programming*. MIT Press 1987.

[Snyder 87] Alan Snyder. "Inheritance and the Development of Encapsulated Software Systems", [Shriver &c 87], 165 ◘ 188.

[Stein 88] Lynn Andrea Stein. "Compound Type Expressions: Flexible Types in Object Oriented Programming". [Meyrowitz 88], 360 ◘ 361.

[Stein &c 89] Lynn Andrea Stein, Henry Lieberman, David Ungar. "A Shared View of Sharing: the Treaty of Orlando". [Kim &c 89a], 31 ◘ 48.

[Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language, Second Edition*. Addison-Wesley 1991.

[Taivalsaari 91] Antero Taivalsaari. *Towards a taxonomy of inheritance mechanisms in object-oriented programming*. Licentiate thesis, University of Jyväskylä 1991.

[Touretzky 86] David S. Touretzky. *The Mathematics of Inheritance Systems*. Morgan Kaufmann 1986.

[Wegner 87] Peter Wegner. "Dimensions of Object-Based Language Design". [Meyrowitz 87], 168 ◘ 182.

[Wegner 90] Peter Wegner. "Concepts and Paradigms of Object-Oriented Programming". *ACM OOPS Messenger* Vol. 1 No. 1 (August 1990), 7 ◘ 87.

[Wegner &c 88] Peter Wegner, Stanley B. Zdonik. "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like". [Gjessing &c 88,] 55 □ 77.

[Whitewater 91] *ACTOR® Programming, Eighth Printing*. The Whitewater Group 1991.

[Yonezawa &c 87a] Akinori Yonezawa, Mario Tokoro (Eds.). *Object-Oriented Concurrent Programming*. MIT Press 1987.

[Yonezawa &c 87b] Akinori Yonezawa, Etsuya Shibayama, Toshihiro Takada, Yasuaki Honda. "Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1". [Yonezawa &c 87a], 55 □ 89.

[Zdonik &c 90] Stanley B. Zdonik, David Maier (Eds.). *Readings in Object-Oriented Database Systems*. Morgan Kaufmann 1990.

[Yu &c 87] C. Yu, C. Van Rijsbergen (Eds.). *Research and Development in Information Retrieval, ACM SIGIR Conference Proceedings.* ACM Press 1987.

CHAPTER 2

ON THE DARKER SIDE OF C++

First published in the Proceedings of **ECOOP '88** (Editors S. Gjessing and K. Nygaard), European Conference on Object-Oriented Programming (Oslo, August 1988), Lecture Notes in Computer Science 322, © Springer-Verlag 1988 (p. 162 - 176). — Reprinted with the permission of Springer-Verlag. This version has more complete trademark acknowledgements. Other differences to the original version are purely typographic.

ON THE DARKER SIDE OF C++

Abstract

We discuss several negative features and properties of the C++ language, some common with C, others pertaining to C++ classes. Remedies are proposed for most of the latter ones, most of the former ones being feared to be already incurable. The worst class-related defects claimed in present C++ have to do with free store management. Some hints are given to programmers on how to avoid pitfalls.

1. Introduction

The C++ programming language [Stro1, Stro2] is a rather new language for which, evidently, no standardising efforts are yet underway; but it has had significant influence on the draft ANSI standard of the C language, as mentioned in [Bana]. It is reportedly used quite a lot at AT&T, where it was originally developed. In addition to that, C++ seems to gain popularity in the UNIX™ community. The USENIX society has recently arranged a workshop on C++ [Caro]. There are commercial implementations available, e.g. [Gloc]. Furthermore, various software packages have been and are being implemented in C++ [Carg, Rich, Wien, Nuut, Gorl]. There does not seem to be much critique yet published on the language; in [Snyd] and [Wegn] some of its features are compared with several other languages. The paper [Nuut] does indicate rather strong discontent with C++, but does not specify it closer (although that paper comes from Finland, too, I do not suspect a general unsuitability of C++ for Finnish temperaments).

This paper tries to bring up some points in and around C++ that I think bad or problematic. Some of them are flaws in the currently available implementations only, some might be considered and ameliorated in the evolution of the language, but some others are certainly inherent and should be taken into account by programmers when deciding which language to use for a given task. The focus will be on semantics, orthogonality, compile-time detection of possible errors, and somewhat on run-time efficiency. Problems that concern concrete syntax only are bypassed, because I consider them both relatively unimportant and to a high degree matters of taste. This paper is *not* a balanced assessment of C++, the language's virtues are mostly mentioned only where they are connected to some problem. In consequence, readers are warned that the language is not as bad as would appear from the present exposition alone; read some of Stroustrup's articles to see the sunny side.

My practical acquaintance with C++ stems from an ongoing project, the purpose of which is a document database management system. In that project, the AT&T C++ Translator, Release 1.0 [AT&T] was used first. Now the work is continuing with Glockenspiel 'designer C++'™ [Gloc], on another computer. While the final revision of this paper was going on, our department got the Release 1.2 of AT&T's product for the first computer; there is no significant experience on it yet. It is a definite lack in my background that I have no practical experience with e.g. Simula™, Ada®, or Smalltalk™. All readers can take § 2 with a grain of salt because I am no authority on object orientation.

At the request of the Program Committee, I have tried to make the text understandable to people without previous knowledge of C++ or even C. For this purpose, there is a short Appendix describing several language features that are used in the examples. Those who are already committed to C will probably either find nothing new in the criticism in § 3 and 4, or disagree with it.

2. About object orientation and language extensions

There is no consensus about what 'object' and 'object orientation' precisely mean. The paper [Stro3] approximately equates 'object-oriented programming' with *'inheritance'*, trying to distinguish it clearly from 'data abstraction', which is presented as another important goal of C++. It looks to me that the issues of object *integrity* and *identity* [Khos] have not been considered important, or that the C heritage has made it impossible to take them very well into account. Interestingly, the taxonomy of [Wegn] also almost ignores the identity and integrity of objects, except in connexion with databases (persistent objects).

Wegner classifies C++ as an object-oriented language, while CLU [Lisk] qualifies as class-based but not object-oriented because there is no inheritance. Ada is classified only as object-based but not class-based because its *packages* have no class, i.e. type. It would be more appropriate to regard the *restricted private types* as classes and their instances as objects; Ada would then be class-based. In the sense of [Wegn], C++ has no data abstraction (because instance variables of objects can be directly accessible) and no virtual resources (because **virtual** functions cannot be left unimplemented in their base class). It has somewhat non-strict inheritance (operations of ancestors can be redefined in descendants, but only if they were declared **virtual** in the first place), and inheritance is by code sharing. It can also be regarded as strongly typed.

A conspicuous omission in [Wegn] is that no distinction is made between languages in which "everything is an object" (Smalltalk-80 [Gold] *et al.*) and those in which objects are just one kind of entity among others. One can write huge programmes in C++ without defining any classes at all. In Smalltalk, one must program in an object-oriented way since no other paradigm is available. This is not to say that the Smalltalk way is "good" — I don't know whether there exists any extremely object-oriented language that offers even nearly the same possibilities for structured programming and compile-time checking as C++ or Ada. Probably the "Turing tar-pit" is easily lurking whenever programming is reduced to a very small set of primitive concepts.

One of the primary goals in the design of C++ was upward compatibility with C [Harb, Bana], as far as feasible. This goal has been very well attained, too. As a consequence, previous C users can quite well upgrade *gradually* to programming in C++, in the first step just feeding their existing C code through the C++ translator and checking if some small modifications would be necessary. Unfortunately, this approach has necessarily transported several drawbacks of C to C++ as well.

If we compare C with Pascal, for instance (a language of roughly the same age), we find that the latter is more object oriented as far as concerns the integrity of data objects. The C language was originally designed with much more concern to machine registers than programmer-defined objects. Moreover, while Pascal is (even overly) strict, C is sloppy. Some features have been defined so as to be convenient for their most obvious application, but causing illogicalities in more complex combinations. (In this respect, C resembles the UNIX command interface.) On the other hand, the existence of pointers to procedures (always 'functions' in C terminology) makes C more object oriented in the sense that behaviour can be connected to data. Further, the generality of pointer expressions can often simplify the handling of complex objects in comparison to Pascal.

Extending some existing language with *lower-level* capabilities is not very difficult in general. The extreme in this direction is the escape to assembler, which exists in several languages or implementations. But when someone sets out to enrich an existing language with object-oriented or other *higher-level* features, trying to keep totally upward compatible with the base language can be problematic. Obviously, it is easier to extend a language that seems too restricted (e.g. Pascal) than one that has very general, powerful, and accordingly error-prone facilities (e.g. C). One recent example of extending Pascal in an object-based direction (in CLU fashion) is presented in [Saje, Olsz].

Several "machine-oriented high-level languages" such as Mary [Conr] have tried to solve the dilemma of powerful low-level features and protected high-level environment by defining a *safe subset* of the language and requiring some explicit operations (e.g. compiler options) or notation for programme modules or sections of them that use *unsafe* features. This principle could perhaps be applied to C++, too, to alleviate the heterogeneity between high-level and even very low-level operations. Of course, it would be much more difficult to decide *a posteriori* which facilities should be classified as unsafe than design a new language with an eye to this classification.

3. Miscellaneous problems inherited from C

The concept of a *type* is somewhat vague already with simple types. For instance, **char** and **short** are something between full-fledged types and **int** crammed into a smaller space. In contrast, e.g. a *pointer* to **short** is a true type of its own, different from a pointer to **int**. Moreover, **long** is a true type, separate from **int**, although they are physically identical in typical 32-bit implementations. Because of operator and function *overloading*, type is a more important concept in C++ than C, and the vagueness thus more irritating. As an example, one cannot define an overloading of a function identifier such that there is one variant for an **int** parameter and another for a **char** parameter.

Enumerations, which possess different degrees of "typeness" in different C implementations (or are not implemented at all) [Harb], are definitely not types in C++, just another way to declare **int** constants and synonyms for '**int**'. This is a pity, especially considering the overloading facility. Furthermore, since C++ has a general means to declare named constants (which C traditionally lacks), enumerations are completely superfluous under the present definition.

A programmer can prescribe the evaluation order of expressions by using parentheses, *except* between operators of the same precedence. The compiler is free to rearrange those operators that are regarded as associative or commutative. This stipulation in C is intended to allow more extensive optimisations. It overlooks the fact that even the basic arithmetic operations are *not* absolutely associative because of overflows, underflows, and rounding errors. This misfeature could be removed from C++ without affecting upward compatibility with C.

The language proper is not concerned with input and output; those functions are relegated to the standard library. (Ada has followed the same model.) This already tends to make them more error-prone than incorporating them into the language, because both compile-time and run-time possibilities to check function parameters are limited (although better in C++ than in most dialects of C). The facilities of the standard I/O functions are on a very low level of abstraction when compared to Pascal. However, just about anything *can* be done using them, whereas standard Pascal I/O is far too restricted for other than toy applications. The low level is probably not a big nuisance to software houses that define and build their own high-level I/O on top — it will be easily (?) *portable* across C++ implementations. — Object input/output is envisaged in [Stro5].

4. Array problems

In my opinion, the worst common feature of C and C++ ("degree of badness weighted by importance") is the handling of *arrays*. An array type, say, *atype* defined by the declaration

```
typedef basetype atype [dimension];
```

is handled as a true type only when storage is allocated for an *atype* variable, or when arithmetic is done in a pointer expression of type *atype** (pointer to array of type *atype*). Otherwise, the name of an array just stands for the address of its first element. Continuing the above example,

```
atype array1, *apointer;
apointer = &array1;
```

the assignment is illegal in C++ (and in many C dialects); it would be legal if *atype* were any other kind of data type except array! (Cf. explanation in appendix.)

Array handling in C and C++ is very much prone to devious programming errors, mainly because it is equated to pointer handling. Of course, indexing *can* be regarded as just a special case of pointer arithmetic, but it is very common, and could be made essentially safer by treating it specially as most languages do. I have not seen "index checking" (what a familiar and natural thing to Fortran and Pascal programmers) mentioned in connection with any C or C++ compiler. There are no aggregate operations for arrays in the language, which means that even assignments between arrays must be programmed by writing explicit loops; this creates more chances for indexing errors.

The original main reason for the unfortunate way of handling arrays was probably a striving to pass parameters and function results in hardware registers. That caused arrays to be passed by address, whereas simple types are passed by value. The principle makes it in most cases impossible to write sensible and efficient array-valued functions. The actual array cannot be a local variable of the function, because it would be destroyed when returning. Therefore, it must be created by the **new** operator, and the caller of the function is responsible for explicitly deleting it later. Actually, the language specification [Stro1] *forbids* array-valued functions, but present compilers accept them gladly.

The unorthogonality of the C and C++ approach to arrays becomes even more evident if we think about embedding an array into a structure with no other components. As a parameter to a function, the structure would then be passed by value, the array by address. Moreover, an assignment statement would be legal for the structure, but not for the embedded array. Conversely, any structure can be embedded in a one-element array, with similar consequences.

A very important special case of arrays are character strings. They also serve well to illustrate the defects of not handling arrays as objects. To make general routines for handling strings of different length at all possible, there is a convention of marking the end of a string with a null byte (the compilers generate it for string literals). This means that the whole string must be traversed even when only the actual length must be found out. Also, there is no built-in way to mark the *reserved* length of a string variable, but the programmer must keep care of it separately. Accordingly, most string handling functions in the standard library come in two variants, one of them having as an additional parameter the maximal number of characters to be read or modified. — Note that there is no way to make a null byte a part of a string as interpreted by the standard functions.

The term 'string' is conventionally used in C and C++ literature to mean a *pointer* to an actual character array. Exactly speaking, the type of such a pointer is not 'pointer to character array' as one would expect, but 'pointer to character' (**char***), i.e. it points to the first character of the actual string. One reason for this convention must be the pointer assignment problem described in the first paragraph. One consequence is that the declared types of a pointer to a null-terminated string and a pointer to a single character become identical.

5. Classes

Classes, borrowed from Simula 67 [Dahl], are the vehicle C++ offers for object orientation. They have facilities comparable to classes in other languages. An equivalent for the **inspect** statement of Simula has deliberately not been included in C++, because it would be contrary to the quest for data abstraction. The possibilities of data hiding are very versatile; they have even been enhanced [Stro4] from the original. Each component of a class is either **private** (the default: accessible only to member and friend functions), **protected** (accessible also to member and friend functions of any derived class), or **public** (accessible wherever the class is defined).

The equivalent of a *method* in Smalltalk is called a *member function* in C++; it is common to all instances of a class. This is quite another thing than a pointer-to-function component, which naturally can be different in different instances. Functions and even whole classes (which means all their member functions) can also be declared **friends** of some class so that they can access its private components. Probably at the time of this conference, available implementations of the language will support even multiple inheritance [Stro4]. Until now, **friend** declarations must often have been used as a substitute for it.

Classes in C++ are defined in such a way that a **struct** becomes just a special case of a **class**, which is nice economy of concept. Also, variables of a **class** type can be defined like any other variables: they can belong to any storage class (need not be allocated by **new**) and be components of arrays and other classes. (Restrictions to this will be mentioned in the following sections.) Class declarations are further organised so as to make a very efficient run-time implementation possible. This principle causes compile-time drawbacks: in most cases, changing the declaration of a class, even the private parts, requires all modules utilising that class to be recompiled [Carg].

Objects of a given class are, in principle, all of the same size. As mentioned in [Stro1] (§ 5.5.8), there are ways to circumvent this restriction for class objects allocated on the free store, but they are not without problems. When one wants to implement classes for things such as well-behaved variable-sized character strings, or anything else of really dynamic size, in practice one has to declare two classes. One of them is the generally visible main class, and the other is an auxiliary class, which contains the actual variable-sized objects and is only used by the main class.

Contrarily to C++, many languages, e.g. CLU [Lisk] and those proposed in [Saje, Olsz], *always* implement aggregates indirectly via implicit pointers, thus increasing run-time overhead for every level of structure in comparison to direct aggregates. The previous paragraphs imply that in several cases, that method can be easier in the programme development phase. To be fair, C++ does not *prevent* a programmer from using classes in an indirect way in order to relieve the compile-time overhead where possible. It suffices to declare

class myclass;

in a source module where only pointers to *myclass* objects will be handled. However, the CLU approach would then result in simpler source code in those modules in which the C++ programmer must be concerned about both *myclass* and *myclass** values. We will return to this subject in § 12. — One can observe that one kind of indirect aggregate is very common in C and C++ programming: the pointer array. It is especially often used instead of an array of character strings, with obvious advantages.

The book [Stro1] uses the word 'member' (of a class) in a meaning that, in my opinion, is in contradiction with its connotations in set theory and everyday speech. In this paper, the word 'component' will be used instead. However, 'member function' does not sound misleading, because normally every invocation of a member function of some class is connected to an object ('member' in the ordinary sense) of the class.

Within a member function, there is always an implicit parameter **this**, which is a pointer to the class instance whose component the function is invoked as. Perhaps a little surprisingly, a member function of *myclass*, say, *can* be called even without an instance of *myclass*:

```
myclass* myc_p = 0;                // null pointer
myc_p -> myfunction ();            // call function via pointer
```

In the above invocation of *myfunction*, the current instance pointer **this** will simply be null. However, this bit of code will crash if *myfunction* is a **virtual** function (cf. § 8), because in that case the class object must really be accessed at run time to find out the appropriate variant of the virtual function. By adding an explicit class prefix like *myclass::myfunction* even a virtual function can be invoked.

6. Problems with constructors and destructors

The possibility to declare constructors for a class is very useful, indeed necessary for achieving sophisticated abstract data types. Among other things, they permit a distinction between initialisation and assignment, which is often crucial (although uninteresting for simple types). Constructors also allow the creation of auxiliary objects "behind the scenes", as mentioned in the previous section. Such constructors are typical cases which necessarily need a destructor as well, to delete the auxiliary objects.

However, constructors and destructors are not without problems, the way they are defined in C++. One obvious defect is that, although constructors will typically take parameters, there is no way to pass parameters in the definition of an *array* of class objects. More subtle difficulties may result from the fact that the order in which the destructors for the automatic variables of a block will be called is undefined.

The capability for the programmer to take care of memory allocation within a constructor is useful; it is the only way to create class objects of variable size. It can also allow a more efficient allocation of specific classes. Unfortunately, it is presently offered in a rather unstructured, "ad hoc" manner, by the appearance of an assignment to the automatically defined variable **this**. (Correspondingly, a zero value can be assigned to **this** in a destructor to bypass standard memory deallocation; this possibility is needed less often.) There is no compile-time check against using such a constructor on external, static, or automatic variables (which are necessarily allocated before the constructor can be called); the programmer must make an appropriate test at run time.

The present C++ translators do not allow the **new** and **delete** operators to be overloaded for a class, contrarily to § 6.2 of [Stro1]. According to [Stro4], the facility will be implemented in the next release. By overloading **new** and **delete** the need to assign to **this** in constructors and destructors can be obviated. Unfortunately, this solution does not work for variable-sized objects.

If one builds a large structure of class objects connected hierarchically (or otherwise) to each other, it can easily happen at the end of the programme that all those objects are destroyed laboriously one by one, to no benefit at all. That can take approximately as much time as building the structure. Fortunately, this "domino effect" can be avoided, e.g. by having enough strategic objects created by **new** and *not* deleting them at the end.

If a class *aclass* has a constructor with *one* parameter of some type *atype* (there can be additional parameters if they have default values), then that constructor will also be used automatically as a conversion function so that

```
atype tom; aclass jerry = tom;
```

will succeed (without compiler warning). This is mentioned in [Stro1], § 6.3.1 and 6.3.2. In many cases, one might not want such automatic conversions, as they can cause programming errors to pass unobserved. Then one must simply avoid defining one-parameter constructors.

7. Mistakes with derived classes

A *derived class* in C++ means a class type that possesses all components of its *base class*, and normally some additional components. A class can also have components of another class type; this is not quite the same as being a derived class, but the problems to be discussed in this section are the same for both cases. The major difficulties with derived classes, and classes with class components, occur in constructor and destructor functions when there is explicit storage allocation and deallocation. That is, we get more complicated problems in addition to those discussed in the previous section.

The "Reference Manual" part of [Stro1] says (in §8.5.5):

> "If a class has a base class or member objects with constructors, their constructors are called before the constructor for the derived class. The constructor for the base class is called first."

Correspondingly, it says (in §8.5.7):

> "The destructor for a base class is executed after the destructor for its derived class. Destructors for member objects are executed after the destructor for the object they are members of."

The reference manual recognises that the case is different if there is explicit storage allocation in the constructor of the derived class, by the following passage (in §8.5.8):

> "Calls to constructors for a base class and for member objects will take place after an assignment to **this**. If a base class's constructor assigns to **this**, the new value will also be used by the derived class's constructor (if any)."

The C++ reference manual errs badly in the last point above: the constructors of both base and derived class should not be allowed to assign to **this**, or conflicting memory allocations will result. (The manual also forgets to say that if a destructor of a derived class assigns a zero value to **this**, then the destructors for the base class and any component classes should be called *before* that assignment.) Even when these errors are corrected, this approach is very difficult in practice, because it cannot generally be known at compile time where the assignment to **this** will actually take place.

At least both C++ implementations mentioned in § 1 make a gross error in the opposite direction to the manual: If there seems to be an assignment to **this** in the constructor (destructor) of the derived class, they simply do not call the constructor (destructor) of the base class at all! This bug must have caused a lot of trouble to people programming in C++. One way of handling the storage allocation problem for derived classes consistently will be presented in § 9.

8. A problem with virtual functions

The smaller difference between the base class of a derived class and a class component of a containing class is that there is no direct way to handle the "base object" as an entity, only its components separately. The main difference is the ability to define **virtual** functions in a base class, which can then be redefined in some derived classes if required.

Unfortunately, virtual functions are another feature, at least in the present implementations of C++, that does not mix freely with programmer-controlled memory allocation. Moreover, neither the book [Stro1] nor the compilers will warn you about the pitfall, which is the following. The "first hook" for the virtual function facility is a pointer, placed immediately after all declared data components of a base class that has at least one member function declared **virtual**. If *variable-sized* objects are allocated, the pointer will be left in the middle. Fortunately, there is a portable way to circumvent the difficulty.

The solution is best illustrated by a small example, showing part of a class declaration (further public components, denoted by the ellipsis, may be functions and variables), a constructor and another public member function. The private member function *contents* is defined within the class declaration itself (thus automatically becoming an **inline** function).

```
class flexstring {
    unsigned space, length;
    char* contents ()
                    // pointer to start of actual string
        { return (char*) this + sizeof (flexstring); }
public:
    flexstring (unsigned = 20);
                    // constructor with default size
    void copy (char*);
                    // copy ordinary string into flexstring
    virtual void put ();
                    // output (somehow)
    . . .
};

flexstring::flexstring (unsigned size = 20)
{
    this = (flexstring*) new char [size + sizeof (flexstring)];
    space = size; length = 0;
}

void flexstring::copy (char* cp)
{
    length = min (strlen (cp), space);
                    // min is not a standard function,
    strncpy (contents (), cp, length);
                    // but strlen and strncpy are standard
}
```

The **sizeof** operator gives the size of a *flexstring* object as known to the compiler, including the virtual function pointer. The constructor allocates space for this *plus* the requested number of bytes for the actual string to be stored. All other member functions that need to access the actual string (*copy* above is a simple example) get its address by calling *contents*.

Another solution to the same problem is to declare an auxiliary class that has a virtual function, then derive *all* classes that need both variable-sized instances and virtual functions from that auxiliary class:

```
class virtual_aid {
    virtual void dummy () { }                    // do nothing
};
```

This solution is simpler (derived classes will not become as contrived as *flexstring* above), but probably has a greater risk of not working with all coming C++ releases if the virtual function mechanisms are changed. The completely straight approach to variable-sized class objects in § 5.5.8 of [Stro1] is successful because the class *char_stack* defined there has no virtual functions.

One should be aware that the **sizeof** operator is purely a compile-time device in all cases. It does not behave at all like virtual functions: if *p* is a pointer to *aclass* then **sizeof***(p*)* will always yield the declared size of an *aclass* instance although *p* may point to an instance of a derived class. It would not even be possible to offer a general "runtime-sizeof" operator without a significant change of current C++ object implementation. This is a consequence of the weak support of object identity.

9. Some suggestions to cope with the problems

We will try to sketch some amendments to the C++ language that would settle most of the difficulties described in § 6 to 8. A complete proposal with a detailed syntax would be a little beyond the scope of this conference paper.

As already mentioned in § 6, it will become possible to overload the **new** and **delete** operators for a class. Very importantly, the **new** operator function will get as one parameter the size of the object to be allocated. It is thus possible to write an allocator for a base class that will work correctly for any derived class also. Alternatively, one may write an overriding allocator for some derived class if needed. In either case, the appropriate version of **new** will be called before any constructor and so the need to assign to **this** within constructors disappears. Hence, the constructors can really be invoked in the order described in § 7.

An analogy of the previous paragraph holds for destructors. However, the **delete** operator function does not get any size parameter. Data structures to store programmer-allocated class objects must therefore be designed so that the size of each allocated object is known at deletion time.

This coming improvement in storage allocation and deallocation will only cater for classes whose all instances are of the same size, as we said in § 6. The reason is that the size passed to **new** is the compile-time (declared) size of the original or derived class. In principle, it would be possible to declare a differing object size for any constructor of a class. This would still be a compile-time matter, thus easily applicable even to automatic, static, and external class variables. With C++ as it stands, a programmer can obtain an equivalent effect by declaring a separate (typically derived) class for each object size. A proliferation of classes can then become a problem, although multiple inheritance may help a little.

An orderly solution to the problem of *run-time* determination of the size of each class instance (cf. example of § 8) would be more complicated. The following is one feasible solution: Corresponding to each constructor for which dynamic size determination is desired, a size calculation function with the same parameter signature as the constructor must be declared. This function will automatically be invoked with the initialisation parameters to yield the size parameter to **new**. After **new** has allocated the correct amount of space, the constructor will be called with the same initialisation parameters as the size calculation function. The compiler should allow such a constructor to be invoked *only* on objects created by the **new** operator. In present C++, the designer of a class has no means to enforce such a constraint, but obeying it is necessary with a class like *flexstring*. I cannot imagine other reasons than the creation of variable-sized objects, that would absolutely forbid a constructor to operate e.g. on automatic variables.

One consequence of the last proposal is that a variable-size constructor must not be used to initialise the base class of a derived class, nor a component of another class. In consequence, the whole example of the previous section cannot be written in this manner by just simplifying the constructor and adding a separate size calculation function: declaring functions **virtual** serves no purpose unless derived classes can be defined. This problem can be solved as suggested in § 5, in a way that might have been clearer in the first place (but we had to illustrate a point in *current* C++): We make the anonymous variable-sized part of *flexstring* a separate class, say *flexbytes*, and add a pointer to it as a component of *flexstring* (the *contents* function is no longer needed). Now, only *flexbytes* has variable-sized instances, and *flexstring* can have virtual functions.

From the object-oriented standpoint, it would appear beneficial if every class instance had a run-time descriptor at its beginning. At present, classes with virtual functions have a kind of descriptor (unfortunately not at the beginning, as explained earlier) but other classes have none. The associated overhead would not be unreasonable even if the descriptor included a length field. A standardised length field would facilitate the writing of constructors, destructors, and memory allocators / deallocators. The **struct** keyword would remain for declaring plain C structures without any implicit overheads.

The two minor problems mentioned in § 6 could be solved if considered worthwhile. Constructor parameters for an array of class objects could be passed by using the same syntax already invented for initialiser lists of aggregates ([Stro1] Reference Manual, §8.6.1). The order of destruction of a block's automatic variables could be defined as the inverse of their creation order; the newer translator versions already seem to work like this.

10. Operator overloading

The capability of operator overloading for class operands is such that a separate function must be written for each desired operator. This can cause some difficulties for both the implementer and the utiliser of a class. The implementer of a typical general-purpose class must write a great number of operator functions. The utiliser must learn the semantics of each operator separately, since they need not have similar relationships to each other as they have with the basic data types.

The most evident area in which the problem just mentioned could be alleviated are the six different relational operators. They could be taken care of by writing only one comparison function, which should be directly accessible, too. Indeed, for comparing *strings*, the standard library of C and C++ contains only a function that returns a negative number if the first string is lexicographically less than the second, a positive number in the converse case, and zero if the strings are equal. An expedient stipulation would be that a comparison function for a class automatically defines all relational operators in the obvious way, but if there is no comparison function then any or all relational operators can be defined explicitly. — The basic idea can be used in defining classes even though it is not built into the language. The paper [Sakk] elaborates on this subject.

The *modifying assignment operators* ('+=', '*=', etc.) are further candidates for reducing the number of functions. Probably the most useful way would *not* be to define them automatically on the basis of the corresponding "ordinary" operators, but *vice versa*. That means, if '+=' is explicitly defined in some class *aclass* for a right-hand operand of type *atype* (not necessarily the same as *aclass*), the variable *a* is of type *aclass* and the variable *b* of type *atype*, then the expression *a* + *b* would be automatically implemented as

```
(aclass temp = a, temp += b)
```

(This is not real C++, since declarations are not allowed within expressions.) An explicit definition of '+' would only be allowed for a class if '+=' is not defined for it (with the same type of right-hand operand). The same principle applies to all modifying assignment operators.

When the operators '++' and '--' are overloaded, the distinction between their postfix and prefix application is lost. This could be remedied, and the semantics of these operators with classes be made even otherwise analogous to that with basic types, as follows. If, for the class *aclass*, the operator '+=' is defined with a right-hand operand of some arithmetic type, and *a* is of type *aclass*, then the pre-increment expression ++*a* would automatically be defined as *a* += *1*. Otherwise, an explicit definition for '++' could be written. The post-increment expression *a*++ would in both cases be automatically implemented as

```
(aclass temp = a, ++a, temp)
```

(Even this is not real C++, of course.) The decrement operator would be handled similarly.

The undesirability of the rearrangement of expression evaluation order, noted in § 3, is really pronounced with overloaded operators. It is totally up to a class implementer to achieve all those commutativity and associativity properties that the compiler assumes some operators to have.

11. Constants and pointers to constants

C++ allows one to derive a constant type from any non-constant data type. This general 'constant' concept is very useful, exists in many other languages, and is unambiguously defined when applied to "pure data" types. However, when the base type is a class with member functions, there arises a problem that none of the references has observed: what member functions, if any, of a constant class instance should be callable? The present implementations appear to allow all member functions to be called, including the assignment operator if it is overloaded. A real solution to this problem would require, either an explicit declaration of those member (and friend) functions that are allowable with constant class objects, or disproportionate run-time effort, at least on typical current computers (with a truly sophisticated hardware architecture like that of Burroughs, it could be easier). We should thus only warn programmers not to trust "constants" of any class that has any modifying member or friend function.

The book [Stro1] discusses pointers to constants (in § 2.4.6). More exactly, a 'pointer to constant' is defined as a pointer through which the referenced object cannot be modified. In consequence, Stroustrup continues:

> "One may assign the address of a variable to a pointer to constant since no harm can come from that. However, the address of a constant cannot be assigned to an unrestricted pointer since this would allow the object's value to be changed."

This is logical. Unfortunately, the old C++ Translator [AT&T] turns things upside down: a pointer to constant can be assigned directly to an unrestricted pointer variable without any warning, but the reverse assignment cannot be done even with an explicit type cast (which normally allows almost anything to be assigned to any variable)! This behaviour has been corrected in the newer releases of the translator.

There are situations in programming when one would like to classify the above kind of pointer as a 'nonmodifying pointer' and have also a 'pointer to true constant' available. Then one could assign a static local *pointer to constant* once in a function and rest assured that even no other part of the programme could modify the constant between invocations of the function. Obviously, assignment of a *pointer to constant* to a *nonmodifying pointer* variable would be allowed, but not vice versa.

12. Some practical difficulties and hints

The definition and implementation of a typical general-purpose class takes quite a lot of effort. Certainly, the same goes for a typical general-purpose private type in Ada. Modules that use several classes will need to include several big header files [Carg] (some of the standard header files needed e.g. for standard I/O are already large). This costs so much in compilation time that one is inclined to write much longer source modules than would be optimal from some other viewpoint. Compilations become expensive and time-consuming also for the reason that the current C++ implementations translate the source code into ordinary C (with an appreciable increase in code size) and then invoke the C compiler. The paper [Dewh] can give interesting insight into some aspects of the language and the AT&T translator, even to persons who are not planning to build their own C++ compiler.

A general guideline for C++ programmers to minimise inclusion and recompilation overheads is as follows: Define hierarchies of derived classes only when every level must be visible to the "end user", as in the example 'employee - manager - director - vice_president - president' ([Stro1], § 7.2.5). Define classes with class components only when needed for runtime efficiency, or if the component classes are very simple. In other cases, declare just pointers to lower-level classes as components of upper-level classes. — Many of the ideas presented in [Stro5] seek to improve essentially the speed and ease of compilation, module management, and software maintenance. When such improvements get implemented, this advice will become less relevant.

Considering the problems discussed in § 8, you should regard any class whose constructor may create instances of different sizes as *abnormal*. Such classes should not be visible to the "end user" but be hidden behind normal classes. An abnormal class should be kept disjoint with all other classes in the sense that it should be neither a derived class itself nor a base class of another, derived class, nor a component of another class. However, no harm can arise from a normal class being a component of an abnormal one. Do not define functions that return abnormal values: according to [Stro6] there is no commitment to support variable-size objects. The earlier version of the C++ Translator [AT&T] handled such return values wrong; current versions handle them correctly in *many* cases, but a function that returns an abnormal class value may again cause mysterious errors with some future release.

If you define some class *aclass* with a constructor (a typical non-trivial class will mostly need it), it is not advisable to define functions returning a value of type *aclass* even if the class is normal. At least current C++ versions implement them in a very inefficient way. It is better to have an additional parameter of type *aclass** or *aclass&* (a reference, cf. appendix), but so that the object to hold the value is created before the call, not in the called function. Likewise for reasons of efficiency, you should avoid passing large class objects directly as parameters, because they must then be physically copied; again, rather use pointers or references, but now *to constant* since the effect of a value parameter is desired. This does not hold if the function really needs a local, modifiable copy or the parameter object. Also, a pointer to an allegedly constant class object is not completely safe (cf. § 11).

To my knowledge, there are at present no debugging facilities usable at the C++ level. When debugging, one must resort to the C level, except for source programme line numbers. The most awkward thing in this is that the C names generated by the translator from overloaded C++ identifiers can be extremely long and hard to type. Moreover, the debugging tools usually available in UNIX environments are rather unsophisticated and hard to use e.g. in comparison to the VAX/VMS™ debugger. Better tools are coming — *Pi* [Carg] is an example of a more advanced debugger. Even today, the personal overall assessment in [Tric] compares C++ favourably over Common Lisp as a software development tool.

13. Conclusion

If a little pun is allowed, perhaps incrementing C by 1 is not enough to make a good object-oriented language to all tastes. The existence of such ambitious object-oriented programming libraries as Gorlen's OOPS [Gorl] is some evidence of the capabilities of C++, although one cannot claim that C++ is object-oriented because OOPS is object-oriented and written in C++. One advantage of C and C++ over several other languages is that the capability to handle many machine-dependent aspects of programming *explicitly* in the language often makes it possible to write very portable code, paradoxical though this may sound. Of course, the same capability also makes it possible for bad programmers to write utterly unportable code.

The realm in which C++ may be competitive against truly high-level, object-oriented languages would be those tasks for which the low-level capabilities afforded by C++ are essential. Development plans for the document database management system mentioned in § 1 are an illustrative example. The user interface layer will probably be realised in Prolog by modifying an existing Prolog prototype [Salm] as required. Modules written in C++ will be used as Prolog primitives. Finally, an existing database management system might be added as the third, lowermost layer.

Acknowledgements

This work was supported by the Academy of Finland and the Ministry of Education (doctoral programme in information technology).

Bjarne Stroustrup (of AT&T Bell Laboratories) has been very helpful and communicative, among other things by sending me copies of some references that would otherwise have been omitted. This was notwithstanding that the first submitted version of this paper, which I sent him, was a lot more arrogant towards his language than the present one. Correspondence with Stroustrup has caused numerous changes especially in § 9 (elsewhere they concern mostly details).

The suggestions of the ECOOP'88 Program Committee have certainly helped to make this paper more generally interesting and understandable than the original version. Seppo Sippu (of our department) has given useful comments at more than one stage of writing.

UNIX is a trademark of AT&T. **Designer C++** is a joint trademark of XEL, Inc. and Glockenspiel, Ltd. **Simula** is a trademark of Simula a.s. **Ada** is a registered trademark of the United States Department of Defense, Ada Joint Program Office. **Smalltalk-80** is a trademark of ParcPlace Systems, Inc. **VAX/VMS** is a trademark of Digital Equipment Corporation.

References

[AT&T] UNIX System V AT&T C++ Translator Release Notes, AT&T 1985 *(307-175 Issue 1).*

[Bana] Mike Banahan, The C Book : Featuring the draft ANSI C Standard, *The Instruction Set Series*, Addison-Wesley 1988.

[Carg] T. A. Cargill, Pi - A Case Study in Object-Oriented Programming, *OOPSLA '86 Proceedings, ACM SIGPLAN Notices Vol. 21 No. 11 (November 1986), p. 350-360.*

[Caro] John Carolan, The Santa Fe Trail, *EUUG Newsletter Vol. 8 No. 1 (Spring 1988), p. 41-44.*

[Conr] Reidar Conradi and Per Holager, MARY Textbook, RUNIT (Trondheim, Norway) 1974.

[Dahl] Ole-Johan Dahl, Bjørn Myhrhaug and Kristen Nygaard, SIMULA 67 Common Base Language, Norwegian Computing Center 1968 *(No. S-2)*.

[Dewh] Stephen C. Dewhurst, Flexible Symbol Table Structures for Compiling C++, *Software - Practice and Experience, Vol. 17 No. 8 (August 1987), p. 503-512*.

[Gloc] designer C++ release 1.2 User Guide, Glockenspiel Ltd. of Dublin 1987.

[Gold] Adele Goldberg and David Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley 1983.

[Gorl] Keith E. Gorlen, An Object-Oriented Class Library for C++ Programs, *Software - Practice and Experience, Vol. 17 No. 12 (December 1987), p. 899-922.*

[Harb] Samuel P. Harbison and Guy L. Steele Jr., C : a Reference Manual, Prentice-Hall 1984.

[Khos] Setrag N. Khoshafian and George P. Copeland, Object Identity, *OOPSLA '86 Proceedings, ACM SIGPLAN Notices Vol. 21 No. 11 (November 1986), p. 406-416.*

[Lisk] Barbara Liskov *et al.*, CLU Reference Manual, *Lecture Notes in Computer Science 114*, Springer-Verlag 1981.

[Nuut] Esko Nuutila *et al.*, XC - A Language for Embedded Rule Based Systems, *ACM SIGPLAN Notices, Vol. 22 No. 9 (September 1987), p. 23-31*.

[Olsz] Jacek Olszewski, Capability Oriented Aliasing Language Rationale, *Technical Report No. 87/89*, Department of Computer Science, Monash University (Australia) 1987.

[Rich] John E. Richards, GKS in C++, *EUUG Newsletter, Vol. 7 No. 1 (1987), p. 53-64.*

[Saje] A. S. M. Sajeev and J. Olszewski, Manipulation of Data Structures Without Pointers, *Information Processing Letters, Vol. 26 No. 3 (November 1987), p. 135-143.*

[Sakk] Markku Sakkinen, Comparison as a Value-yielding Operation, *ACM SIGPLAN Notices, Vol. 22 No. 8 (August 1987), p. 105-110.*

[Salm] Airi Salminen, A method for designing tools for information retrieval from documents, *Proc. 4th Symp. on Empirical Foundations of Information and Software Sciences (1986), p. 261-272*, Plenum Press 1988.

[Snyd] Alan Snyder, Encapsulation and Inheritance in Object-Oriented Programming Languages, *OOPSLA '86 Proceedings, ACM SIGPLAN Notices Vol. 21 No. 11 (November 1986), p. 38-45.*

[Stro1] Bjarne Stroustrup, The C++ Programming Language, Addison-Wesley 1986.

[Stro2] Bjarne Stroustrup, An Overview of C++, *Object-Oriented Programming Workshop, ACM SIGPLAN Notices Vol. 21 No. 10 (October 1986), p. 7-18.*

[Stro3] Bjarne Stroustrup, What is "Object-Oriented Programming"?, *Proc. 1st European Conf. on Object Oriented Programming, Paris (June 1987)*, also to appear in *IEEE Software, May 1988*.

[Stro4] Bjarne Stroustrup, The Evolution of C++ : 1985 to 1987, *Proc. USENIX C++ Workshop, Santa Fe, New Mexico, U.S.A. (November 1987).*

[Stro5] Bjarne Stroustrup, Possible Directions for C++, *Proc. USENIX C++ Workshop, Santa Fe, New Mexico, U.S.A. (November 1987).*

[Stro6] Bjarne Stroustrup, *private communication*, 1988.

[Tric] Howard Trickey, C++ versus Lisp: A Case Study, *ACM SIGPLAN Notices, Vol. 23 No. 6 (February 1988), p. 9-18.*

[Wegn] Peter Wegner, Dimensions of Object-Based Language Design, *OOPSLA '87 Proceedings, ACM SIGPLAN Notices Vol. 22 No. 12 (December 1987), p. 168-182.*

[Wien] Richard S. Wiener, Object-Oriented Programming in C++ - A Case study, *ACM SIGPLAN Notices, Vol. 22 No. 6 (June 1987), p. 59-68.*

Appendix: Some features of C++ (and C)

The languages C and C++ can be called Algol-like: they are imperative and block-structured, have largely the same complement of statement types and basic data types as any language of the Algol family, and allow recursion. Conspicuous syntactic differences from Algol 60 are an easier attitude to semicolons and the substitution of **begin** and **end** by '{' and '}' respectively. Comments in C are bracketed by '/*' and '*/'; C++ additionally allows end-of-line comments beginning with '//'.

C (and C++) has a text-substitution preprocessor facility that allows macros both with and without parameters. The capabilities most often needed in C++ are probably compile-time inclusion of secondary source files and conditional compilation. Several other things, common in C (e.g. defining symbolic constants as macros), can be better done in the C++ language proper. The preprocessor is rudimentary in comparison to any modern macro assembler.

The fundamental data types *of* C are signed and unsigned integers of several sizes (**char** is most naturally considered one of them), floating-point numbers, and **void** (an empty set of values). The most important derived data types are arrays (many-dimensional arrays are handled similarly to Pascal), structures (**struct**, like records in Pascal), and pointers. The *classes* of C++ are discussed in the main text. C++ allows the definition of constants (**const**) of any type.

The logic of a type definition is approximately inverse to that in most Algol-like languages: it tries to describe how one will get a value of the base type from the declared variable (or, in the case of a **typedef** statement, from a value of the new type). Thus, in the example of § 4, indexing an *atype* value gives a *basetype* value, the variable *array1* is of *atype*, and dereferencing *apointer* ('*' is the dereferencing or indirect addressing operator, prefixed to its operand) gives an *atype* value. The type 'pointer to *atype*' is often denoted by '*atype**'. The unary, prefix '&' is the referencing or address-of operator: when applied to an (addressable) object of type *sometype*, it yields a value of type *sometype**. The last statement in the example is erroneous because *array1* is not regarded as a variable of type *atype*, but instead as a constant of type *basetype**, namely *&array1[0]* (array indexes always start from 0).

Function declarations follow the same logic as variable declarations. In fact, the only thing that distinguishes the declaration of an *atype* variable from the declaration of a function returning *atype* is that there must be a pair of parentheses after the function name (even if there are no formal parameters). A function declaration that is also a *definition* is recognised from a block of code (in braces) immediately following the parameter parentheses. C++ accepts the attribute **inline** for a function, meaning that the function body shall be in-line expanded at every place where the function is called, thus minimising the overhead for very small functions.

Components of classes (both data and functions) are accessed by conventional dot notation. Alternatively, a right arrow can be used in conjunction with a *pointer* to the class type (as in the example of § 5); this is not essential but handy because the dereferencing operator '*' has lower priority than the dot (component selector).

Typing is strong as a rule (cf. § 3), but there are some implicit type conversions (cf. example in § 6). Furthermore, explicit type conversions or casts can be effected very generally between types; even if *atype* cannot be converted to *btype*, at least *atype** can be converted to *btype**. The example of § 8 uses traditional C cast notation in the constructor function *flexstring::flexstring* to convert a **char*** value to *flexstring**. Whenever the target type can be expressed as just a type name, functional notation can be used as well:

```
this = flex_pointer (new char[somesize]);
                        // suppose type flex_pointer has been defined
```

In addition to pointers, *references* to any data type can be declared in C++: *atype&*. A reference is semantically almost the same as a constant pointer but can cause the automatic creation of a temporary variable to refer to, in some cases. Syntactically, declaring a formal parameter of a function to be *atype&* (instead of *atype**) makes function calls look just as if one had a reference parameter (**var** parameter in Pascal).

The most important storage classes of variables are **extern** (global), **static** (roughly equivalent to **own** in Algol 60), and **automatic** (allocated on the stack). The allocation and deallocation of variables in free store (dynamic memory, heap) is similar to Pascal. C++ has the standard operators **new** and **delete** for this purpose; in C various library functions are used (depending on the environment and implementation).

C and C++ are statement languages, not expression languages like Algol 68. However, instead of the more conventional assignment statement, the main workhorse is an *expression statement*. The assignment operator '=' (not to be confounded with the equality test operator '==') is just an operator that both has a side effect and yields a value. In addition to ordinary assignment, there are "modifying assignment" operators corresponding to most binary operators: their left-hand operand is evaluated once, then used in the binary operation, and last assigned the result of the operation (cf. § 10). As a special case, there are *unary* operators '++' and '--' for incrementing and decrementing an arithmetic variable by one. They can be used as either prefix and postfix operators; the value of the postfix expression is the *old* value of the variable. Another unconventional operator, usable within any expression, is the sequencing operator ',' — its left-hand operand is evaluated first (presumably for the sake of its side effects) and its value discarded, the right-hand operand is evaluated then and its value used for the whole expression (cf. § 10).

There is no 'main programme' nor are there 'procedures' in C or C++; all *statements* are within *functions*. A function with the name *main* will be recognised as the main programme, and functions with result type **void** (thus returning no result) can be defined. *Any* function can be invoked in the manner of a procedure call if the result is not needed. Functions cannot be lexically nested; all functions are either on the global level or within class declarations. It is possible to define **extern** and **static** variables outside of functions. The type of a function is defined by its signature, i.e. the type of result and the number and types of formal parameters; traditionally in C the type of a function has been determined solely by its result type. (We use the word 'parameter' in this paper, although the C and C++ community prefers 'argument'.)

Code and data are completely separated in principle, but pointers to functions are possible. There is a distinct pointer type corresponding to every distinct function type. If an implementation does not completely protect code segments at run time, code can naturally be mutilated, e.g. after casting a pointer to a function to another pointer type.

The programmer can declare any function name to be overloaded; names of member functions (of classes) are automatically regarded as overloaded. The C++ translator determines the correct function to apply from the types of actual parameters and result, applying standard conversions if necessary. Overloaded functions with the same name must therefore be distinguishable from each other by their signatures in the C++ type system. The class of a member function can be explicitly specified in a function call thus (the type of *anobject* must be *thatclass* or derived from it):

```
z = anobject.thatclass::somefunction (x, y);
```

Virtual functions of derived classes will very often need this possibility to call the corresponding functions of their base classes. — Almost all *operators* can be overloaded analogously to functions.

It is possible to define a *constructor* function for a class, even several constructors if they have different parameter signatures. If this is done then it is guaranteed that any instance of the class, independently of storage class, will be automatically initialised by the appropriate constructor before first use. Likewise, it is possible to define a *destructor* function (only one) for a class. In this case, the destructor is guaranteed to be called automatically to operate on every instance of the class when it is being deleted. Note that this also happens to external and static variables of a class type at programme exit, but not to instances created by the **new** operator unless they are explicitly deleted by **delete**.

CHAPTER 3

COMMENTS ON "THE LAW OF DEMETER" AND C++

First published in ACM SIGPLAN Notices, Vol. 23 No. 12 (December 1988), p. 38 - 44. — Reprinted according to the policies of SIGPLAN (copyright not transferred to the publisher).

COMMENTS ON "THE LAW OF DEMETER" AND C++[1]

Abstract

A rule of good style for object-oriented programming has recently been put forward, actually in several flavours (class vs. object, weak vs. strong). Some possible problems in the original rule are discussed, and a modified formulation is proposed to overcome at least part of the them. Doubts still remain about how useful the rule is with Smalltalk and other *untyped* languages. Then the application of the rule to the C++ language is studied and shown not to be as straightforward as has been suggested. This is largely a consequence of the intertwining of the conventional and the object-oriented component in C++. On the other hand, being typed, it is a promising language for enforcing rules of this kind at compile time. A new language-specific formulation is finally presented, argumenting that it is better in class than object flavour.

[1] This work was supported by the Academy of Finland.

1. Introduction

The paper [Lieb2] (a shorter version appeared as [Lieb1]) suggests a simple rule for writing "good" code in an object-oriented language, analogous to the rules of structured programming for conventional languages. This rule is called "the Law of Demeter" or "the Law of Good Style". Its essence could be put shortly thus (my suggestion): **The methods of a class should not depend in any way on the structure of any class, except the immediate (top-level) structure of their own class. Further, each method should send messages to objects belonging to a very limited set of classes only.** Together with the Law itself, the authors recommend programmers to *minimise:*
(M1) code duplication,
(M2) the number of arguments passed to methods,
(M3) the number of methods per class.

The advantages of following all these guidelines are classified into six areas in [Lieb2] §4:

- coupling control,
- information hiding,
- information restriction,
- localisation of information,
- narrow interfaces,
- structural induction.

When one speaks about programming *style*, that normally includes only properties that can be assessed by static analysis of source code. This has been the starting point in the original papers on the Law of Demeter, of course. We will introduce here a new distinction to make our reasonings clearer: given a style rule, we will say that a programme (or module, or fragment) is:

- in *good* style if it can be verified by static analysis to obey the rule;
- in *bad* style if it can be verified by static analysis to break the rule;
- in *questionable* style otherwise.

Evidently, a style rule is the less useful, the more code fragments it will classify as questionable; although a rule that makes everything good or everything bad is worthless as well.

The present article presupposes that readers have either of the original papers available, preferably [Lieb2]. Otherwise it would have been necessary to quote too large parts of them. Those papers, although trying to be independent of any particular language, were written somewhat from the viewpoint of Lisp/Flavors, and I am not familiar with it. That might have caused some slight misunderstanding of the law's consequences in the next section. I hope that the terminology differences between the literatures on different languages will not create confusion.

Both in the original paper on the Law of Demeter and in the present one, there is a danger of the main message being submerged under a mass of technical detail. We recommend the reader to keep an eye on the bold-printed passage in the first paragraph above.

2. The law in class and object forms

The Law of Demeter is originally formulated in [Lieb2] §3 for the Demeter™[2] interface built on top of Lisp/Old-Flavors; while Flavors is an untyped language, Demeter seems to be *typed*. The law goes effectively as follows:

(A) For all classes C, and for all methods M attached to C, all objects to which M sends a message must be instances of classes associated with the following classes: (read simply: "... must be instances of the following classes:" for Lisp/Flavors without Demeter)
(A1) The argument classes of M (including C).
(A2) The classes of objects created by M, or by functions or methods which M calls.
(A3) The classes of global variables.
(A4) The instance variable classes of C.

In [Lieb1] and [Lieb2] the cases (A2) and (A3) are included in (A1) by saying that "[those objects] are considered as arguments of M". We have paraphrased the law here in a more systematic way. Obviously, we should then, in §1 of this paper, modify item (M2) in the list of things to be minimised into:
(M2´) the number of global variables, arguments passed to methods, and object creations (direct or indirect) in methods;

In §7 of [Lieb2], one special circumstance is described that "violates the spirit of the Law": In many OOPL's including Lisp-Flavors-Demeter, every instance variable of a class causes a method of the same

[2] Demeter is designated as a trademark in [Lieb2].

name, returning the current value of the instance variable, to be generated automatically. (Another method for *setting* the value of the instance variable is generated likewise.) Invoking such methods effectively penetrates one level into the structure of the objects concerned. We can assume that the authors want to implicitly follow an additional restriction that would be approximately as follows:

(A5) No message that directly names an instance variable of its receiver must be sent unless that instance variable's class is itself covered by some of the conditions (A1) - (A4).

The notion of instance variables in (A4) above is ambiguous on purpose. In §8 of [Lieb2] a distinction is made between the Weak and the Strong Law of Demeter, depending on whether *inherited* instance variables are included or not. As noted there, this distinction is not needed with languages in which the visibility of instance variables can be defined in class declarations. This happens to be the case in newer versions of C++ [Stro2] (as correctly mentioned in [Lieb2], too): they have the choice of visibility attributes *public*, *private*, and *protected* (visible to class itself and its subclasses).

The above rules (A) are stated purely in terms of classes (types). The Law of Demeter is later sharpened into terms of objects, motivated by an example that is regarded as pathological ([Lieb2] §4). The object form can be formulated thus:

(B) For all classes C, and for all methods M attached to C, all objects to which M sends a message must be:

(B1) M's argument objects (including the *self* object), or
(B2) objects created by M, or by functions or methods which M calls, or
(B3) objects in global variables, or
(B4) instance variable objects of C.

The "implicit rule" (A5) would naturally be transformed into:

(B5) No message that directly names an instance variable of its receiver must be sent unless that instance variable's object is itself covered by some of the conditions (B1) - (B4).

The paper [Lieb2] claims that the object form is more natural from the conceptual point of view, but harder to check at compilation time than the class form. It concludes that it is better to retain the class form, also because pathological cases as in the example are not likely to occur often.

3. Critique of the law in general

Concerning the choice between class and object formulation, I would make a stronger statement than that in the previous paragraph. The Law of Demeter is concerned with *style*, i.e. static reasoning about source code (cf. §1). In a language with sufficiently strict type checking, every statement can be classified as being in either *good* or *bad* style according to (A) in a given context ([Lieb2] §10). If we switch to the object form (B), then most *bad* statements will become *questionable* style, because only in few special cases (e.g. literal constant objects) can two distinct object references be compile-time verified to certainly denote different objects. No bad statements can become good, but some good statements will become bad. Some good statements will also become questionable style — probably often a large proportion of them. This means that the object law is a poorer style guide than the class law.

Going to the object form would tend to mix the main issue of the Law of Demeter with some rather different objectives. The "pathological" cases remedied by the object formulation can be seen as run-time problems, and they violate only one of the six principles listed in §1, namely information hiding. One can construct examples that are somewhat similarly pathological but yet do not violate even the object form of the law. The situation is analogous to the fact that the rules of structured programming cannot remove all anomalies from procedural programming. In my opinion, we should not try to accomplish too much at one scoop.

Such skepticism notwithstanding, the object formulation *is* in some way more satisfying conceptually. The class formulation will in most cases allow each method invocation to access a very large set of objects it should logically better not deal with. To subdivide classes only in order to restrict the accessibility of objects to various methods would be in glaring contradiction to the Razor of Occam ("do not multiply concepts without necessity"). One can take the object Law as a goal to be approximated in programming; the main objection above was that it is not enforceable or checkable.

As a more technical remark: in order for the object form (B) of the Law to be *really* in terms of objects, obviously one should restrict alternative (B4) into either one of the following:

(B4´) instance variable objects of those of the objects mentioned in (B1) - (B3) that belong to class C.

(B4´´) instance variable objects of the *self* object.

The rule (B5) should be restricted respectively into either one of the following:

(B5´) No message that directly names an instance variable of its receiver must be sent to other objects than those mentioned in (B1) - (B3) that belong to class C.

(B5´´) No message that directly names an instance variable of its receiver must be sent to other objects than the *self* object.

The rest of this section concerns the class and the object form equally.

The paper [Lieb2] contains *no motivation* explicitly for (A2) and (B2). Including them seems to run counter to the purposes of the Law, viz. to simplify modifications and to simplify the complexity of programming. Especially allowing messages even to classes of objects created by methods which M calls, makes it possible to look more than one level deep into the structure of objects, which the Law specifically tries to banish. — On the other hand, probably the *result* class of M should be included in the list, even in the case that M does not *create* its result object.

No explicit motivation is given for (A3) and (B3), either. In fact, I don't know exactly what 'global variables' mean in this context. It seems reasonable that standard built-in classes such as *integer* should be allowed, at least in those languages in which their definitions are immutable. With regard to the goals of the Law (§1), I would still like to require in the heading of class C an explicit listing of all classes allowed under (A3) in C's methods (*'known classes'*). This would be a restriction of the original law; on the other hand, we could get a useful *relaxation* of the law by allowing *any* class K to be declared known to C, or C an *acquaintance* of K. We will speak more about acquaintances in §7.

The explanatory text in the beginning of §4 of [Lieb2] stresses that the immediate structure of C but not of M's arguments is allowed to be known by M — rule (A4). This keeps us unnecessarily trapped within the unelegant asymmetry of most OOPL's that the first operand of an operation is in a very different role (the object owning the method) than all others (the arguments). One elegant exception is CLOS [Bobr]; C++ is another exception, and we will examine it closely in the later sections. I suggest that the instance variable classes of the argument and result classes of C should also be allowed. For a nice short justification of this, see e.g. [Øste].

4. Applying the law to untyped languages

The majority of presently popular OOPL's, including Smalltalk-80™[3], Objective-C™[4] (its *object* component), and CLOS, are *untyped*. As §10 of [Lieb2] shows, the authors have pondered on the difference between typed and untyped languages, but maybe not sufficiently. They describe three different cases that make "compile-time" checking of the Law of Demeter impossible (i.e. make pieces of code to be in questionable style), but fail to mention explicitly the most common and most important reason: that the types (classes) of objects owned by variables cannot be known at compile time. This will certainly cause a large part of normal code written in any untyped OOPL to be in *questionable* style (§1).

Let us compare the class and object forms of the Law of Demeter as applied to an untyped language. I suppose that object identity is never easier to decide at compile time than class equality, in most languages. (However, one could possibly devise a language construct in which two objects could be verified to be different but the equality on their classes would be undecidable.) As long as this assumption holds, all code that is questionable under the class form will stay questionable under the object form; thus the object law cannot be better than the class law even for an untyped language. But we will see that, contrarily to the situation with a typed language, it is no worse either.

The only case in which we can be sure of an object being of the same class as an argument of the method M is referencing the object via that argument; hence substituting rule (A1) by rule (B1) makes no good fragment into questionable or bad. Conversely, there is *no* case in which we can be sure of an object *not* being of the same class as an argument; hence no fragment is bad on the basis of rule (A1) and, vacuously, no bad fragment can become questionable by introducing rule (B1). Thus, rule (B1) is no worse than rule (A1). — We can handle the rest of the rules pairwise in the same way (taking some reasonable interpretation of 'global variables' in (A3) and (B3)) and finally conclude that the object form of the Law of Demeter is no worse than the class form, for an untyped language.

Unfortunately, the main lesson from the previous reasonings is that **the Law of Demeter is not very effective with untyped languages.** This can be seen either as a defect in the law or as yet another argument

[3] Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

[4] Objective-C is a trademark of The Stepstone Corporation.

in favour of strong typing in OOPL's, or both. I would draw mainly the second conclusion: part of the price languages such as Smalltalk must pay for their dynamism and malleability is that the possibilities for static analysis *are* very limited. Perhaps we can imagine some amount of run-time checking of the Law in the test phase of new software; an end user would probably not be interested to get sporadic error messages like "Method *M1* of class *C1* broke the Law of Demeter by sending message *M2* to an object of class *C2*". As in all testing and debugging, only negative evidence from such run-time checking would be certain. Probably run-time checking is a bit more expensive for the *class* form of the Law — one must go from each object to its class and then look for equalities — so that the *object* form could be preferred.

The paper [Lieb2] at many places proposes rather a reasonable aspiration than absolute conformance to the Law of Demeter, as witnessed by its §11 on minimum documentation. From this point of view, neither the object form of the law nor its application to untyped languages are so ineffective as from the more formal perspective we have taken here.

Contrarily to their conclusion mentioned at the end of the previous section (at the end of §4 of [Lieb2]), the authors have chosen the *object* form of the law in the formulations for other OOPL's in §12 of [Lieb2] — perhaps motivated by problems such as above with untyped languages. These reformulations have unfortunately been too hastily written in general, without really considering all relevant differences of the languages. For instance, concerning Smalltalk-80, messages to the *class* of C (i.e. a metaclass object) are not explicitly mentioned, although they should surely be allowed; the same holds for the (meta)classes of all other classes covered by (B1) - (B4).

5. A new suggestion

In order to be a little more constructive, we present a complete new version of the Law; as a consequence of our previous reasoning it is in terms of classes, not objects.

(C) For any class C, and for any method M attached to C, every object to which M sends a message *must* be either one of the following classes or an instance of it, in order of *decreasing preference*:

(C1) the class C itself;
(C2) the classes of the arguments and result of M;
(C3) the instance variable classes of C;
(C4) the classes explicitly declared as known to C;
(C5) the instance variable classes of the classes mentioned in (C2).
It is additionally required that
(C6) *no* message that directly names an instance variable of its receiver *must* be sent, unless the receiver is an instance of one of the classes mentioned in (C1) and (C2).

The rule (C6) would be stronger and perhaps better if modified thus:
(C6´) *no* message that directly names an instance variable of its receiver *must* be sent.
However, I feel that this would be implicitly in terms of objects, while the form (C6) is strictly in terms of classes.

To the list of things to be minimised (in §1), we should add the following:
(M4) the number of all classes in a system;
(M5) the number of classes to which and to whose instances messages are sent, both per method and per class (union over all methods);
(M6) the number of classes covered by *both* (C1) or (C2) or (C4) *and* (C3) or (C5), for each method;
(M7) the number of known classes per class.

Note that the items (M4) - (M6) are equally valid for the original law (**A**), except that (M6) would need an accordingly different formulation:
(M6´) the number of classes covered by *both* (A1) or (A2) or (A3) *and* (A4), for each method;
This item tries to further prevent the same kind of anomalies as rule (A5) or (C6), without resorting to a formulation in terms of objects. Note also that item (M2) from §1 is valid in its original form for our formulation (**C**), (M2´) from §2 is not needed.

We will try also an *object* formulation, equivalent to (**C**) as far as possible. The most problematic detail in this translation is (C4), because we really want something more restricted than all global objects (B3). Now we do not try to define any order of decreasing preference, because the rule would become too complicated.

(D) For any class C, and for any method M attached to C, every object to which M sends a message *must* be either one of the following objects or its class:

(D1) the arguments and result of M and the *self* object;
(D2) objects owned by global variables or created by M, which additionally belong to the same class as at least one object mentioned in (D1);
(D3) the instance variable objects of the objects mentioned in (D1) and (D2);
(D4) objects owned by global variables or created by M, which additionally belong to a class explicitly declared as known to C;

It is additionally required that

(D5) *no* message that directly names an instance variable of its receiver *must* be sent, except to the objects mentioned in (D1) and (D2).

Both 'global variables' and 'created by M' remain somewhat vague in this language-independent formulation. The adaptation of rules (D2) and (D4) to specific languages thus requires some reflection.

6. Interpretation of the law for C++

C++ is essentially a strongly typed language (cf. [Sakk] for some flaws in its type system). The philosophy of C++ includes a much more static binding strategy (e.g. between functions and their argument types) than is typical in OOPL's. Most importantly, every invocation of a member function is closely bound to a particular class type. Thanks to this, it would be possible to enforce the Law of Demeter in its class form more easily and completely at compilation time than in most other languages. For that purpose, one would need a modified C++ compiler (or translator), most conveniently with a Demeter option, which could be omitted in order to accept normal (unrestricted) C++.

The *object* version of the Law of Demeter is reformulated in §12 of [Lieb2] for C++, although the class version could have been preferred. The law is effectively as follows (paraphrased again):

(E) For all classes C, and for all member functions M attached to C, all objects to which M sends a message must be either:

(E1) M's argument objects, including the object pointed to by *this*.
(E2) Objects created by M, or by functions which M calls.

(E3) Objects in global variables.
(E4) Data member objects of class C.

In the above, the translations of 'method' to 'member function' and 'self' to 'this' are probably the best possible, and the translation of 'instance variable' to 'data member' is almost correct (but see next paragraph). Yet, the adaptation of the law's original formulation to C++ terms is badly incomplete. It must still be elaborated quite a bit, mainly because C++ is not a totally object-oriented language [Sakk], but partly also because of additional terminology differences. To start with, 'object' in the Law is used in the OOP connotation, but in C++ an object is only "a region of storage" ([Stro1] §r.5). No messages can be sent to objects, in fact there is no 'message' in the whole C++ terminology. In a way, 'to send a message to an object' would be approximately translated into 'to invoke a member function of a class object', but we will in fact equate it with 'to access a class object'.

We should note that a class is not an object in C++, but it is possible to declare "static data members", which are equivalent to class variables. The C++ formulation of the Law of Demeter as given above, clearly implies that static members can be accessed, too. Probably this *was* the authors' intention, but if it was not, then the formulation should be amended accordingly. Then we should also either forbid static data members altogether or devise an equivalent of *class methods* to handle them. This is possible by declaring *friend* functions to the class. By the way, the explicit or implicit invocation of a C++ *constructor* is very much analogous to the invocation of a class method.

As we said in §2, the roles of the owning object and the explicit arguments of a function are definitely more symmetric in C++ than in most other OOPL's. In the special case of overloaded binary operators, the syntax is even more symmetric than the semantics: you can write 'obj1 * obj2 + obj3'. In other cases, you must write e.g. 'obj1.memfunc (obj2, obj3)', but the particular variant of an overloaded function to invoke is always decided on the basis of the types of *all* arguments (and the required result type, if it can be seen from the context). This is done at compile time (as a contrast, CLOS does essentially the same thing by run-time dispatching), but a more specific variant (with the same types of result and explicit arguments) can be selected at run time if the function is *virtual*.

Whenever a truly symmetric function needing to access components of objects belonging to several classes is desired, it can be defined as a friend of all those classes but a member of none. Here we have another situation in which friend functions are useful. Accessing static data members was the first such situation; in a case in which we need *only* access static members and no instance of the class at all, using a *member* function would be unnatural.

7. A new proposal for C++

Before going deeper into the details of the C++ language, we present a new version of the Law of Demeter for C++, derived from the form (**C**) in §5. It is in terms of classes, and should in principle be enforceable at compile time.

(F) For any class C, and for any member function M of C, every class object that M directly accesses or references in any way *must* belong to one of the following classes, in order of *decreasing preference*:
(F1) the class C itself;
(F2) the classes of the arguments and result of M;
(F3) the data member classes of C;
(F4) the classes explicitly declared to be known to C;
(F5) the data member classes of the classes mentioned in (F2).
Further, this access or reference
(F6) *must not* directly specify any *data* member, class or non-class, of the object concerned unless the object is an instance of one of the classes mentioned in (F1) and (F2).
For any function M that is *not* a member of any class, the above rules shall hold with the addition that classes can be explicitly declared known to M.

Let us look at the most important implications of (**F**). First, we must clarify a little what we mean by the function M 'directly accessing or referencing' an object of a given class, say K. That means the appearance of any expression of type K in the code of the function M itself, not in another function that M calls. Also, the mere appearance of an object of type *pointer* or *reference* to K does not count. 'Directly specifying a data member' (F6) means that either the direct ('.') or indirect ('->') member selector with the name of a data member appears in the code of the function M itself.

Second, the last paragraph of (**F**) is necessary because C++ functions need not be members of any class. For such functions, the alternatives (F1), (F3), and (F4) are obviously vacuous.

Third, we will not make writing law-conforming programmes essentially more difficult, but will make checking much easier, by demanding that no *data* member of any class be declared *public*. This will prohibit most illegal accesses. To allow legal accesses belonging to case (F5), the function M needs to be declared a *friend* of its result and argument classes. This declaration must be inside the definitions of those classes; the situation is thus very similar to M being a *member* function of several classes simultaneously. This alternative should not be exploited without true need, therefore it can well be somewhat laborious. We can also prohibit all friend declarations except those just described.

Fourth, in order to utilise alternative (F4), a programmer will need to declare *acquaintances*. This is the only addition to standard C++ syntax required by the present suggestion. While friends of a class K are declared in the definition of K, class C being an acquaintance of K is declared by declaring K *known* in the definition of C. This is a strictly weaker condition than class C being a friend of K: the member functions of an acquaintance can only access an object of class K as a whole and invoke its *public* member functions; the member functions of a friend class can access f2all function and data members of a K object.

8. Non-class objects and the Law

None of the built-in datatypes of C++ (essentially the same as in C) is a class, and more *non-class* data types can be derived from existing class or non-class types also as vectors (arrays) and pointers, even pointers to functions being possible. (Unfortunately, C++ nomenclature has reserved the term 'derived class' to mean 'subclass', so we must be cautious with the word 'derived' to avoid ambiguities.) We will try first the most straightforward approach: that the Law of Demeter *shall not concern* non-class objects.

The class concept of C++ is intentionally defined so as to comprise the structures (*struct*) and *unions* of C as well. For the sake of simplicity, we leave unions out of the discussion; their utility as classes is very limited anyway. For the purposes of the Law of Demeter we will demand a clear dichotomy: only "pure structures" and "pure classes" shall be allowed, no intermediate forms, which are possible in C++ as such. Classes shall be defined with the keyword '*class*' and have no public data members (as already required in the previous section). Structures shall be defined with the keyword '*struct*'; their all data members shall be public, and they shall have no member functions, probably no static members either. Structures will then be considered non-class aggregations like vectors.

The consequences of this approach to the interpretation of rules (F3), (F5), and even (F6) are not quite self-evident: what exactly are the data member classes of some class C? There is no problem when each data member is either of another class type or of a "purely non-class" type. In "mixed" cases, e.g. when a data member of class C is a vector of pointers to class objects (of class D, say), that class D should be regarded as a data member class of C.

The principle described in the previous paragraphs would tend to minimise the number of distinct classes in a hierarchy. A diametrically opposite approach, maximising the number of classes and also the encapsulation, could be the following:

(N1) Regard as a class type any type that is a class itself (pure class) or derived (even partially) from a class type. A function type is regarded to be derived only from its result type, not the argument types.

(N2) The data members of a class must be either all of non-class types or all of class types.

(N3) If a class has more than one data member of class type, each of them must be a pure class.

(N4) If a class has only one data member of class type, it can also be a vector of pure class objects, a pointer or reference to a pure class, or a pointer or reference to a function returning a class value.

The same distinction between classes and structures that was made in the beginning of this section is understood. The above rules will allow structures of pure non-class objects only. — The complexity of these definitions stems mostly from the fact that the abstraction and encapsulation facilities of a class cannot be used in C++ with other type derivations than *class* = *struct* (and *union*). This is in contrast to e.g. Demeter ([Lieb2] §2), although Demeter does not offer as many derivation methods as C++, only construction (aggregation), alternation (union), and repetition (list aggregation).

The alternative approach could be pushed even further by considering every built-in datatype a pure class. (E.g. in CLOS, a corresponding class has been defined for every Common LISP data type.) The rules (1) and (2) in the previous paragraph would then be vacuous, and *every* derived type should in effect be a class, too.

9. Further subtleties with C++

Member functions of classes can be either private, protected, or public; this facility of C++ can be employed by a programmer to define additional restrictions beyond the Law of Demeter.

Explicit pointers are essential types in C++, and pointers to class objects are necessary to implement any conventional dynamic data structures. This brings us all the familiar problems with null pointers, dangling references, multiple references, etc., but such dynamic problems were not the issue of the Law in the first place. Under the interpretation of the previous section, pointers can e.g. cause a class object to be simultaneously the object owned by one or more of its own instance variables. I suppose even this is not contrary to the spirit of the Law, at least if we stick to the class formulation.

An important positive factor is that C++ pointers are typed. However, the language allows explicit type casts between arbitrary pointer types. In fact, a perverse programmer can pretend any type of data or function to reside at any place in the available address space (subject only to the alignment rules of the machine). If we really tried to adapt the object form of the Law of Demeter to C++, we should prohibit such type casts, and probably some other circumventions of the type system as well. Pointer *arithmetic* and even array indexing are unsafe operations: even they easily allow a part of memory to be accessed as if it contained a certain type of data, no matter what there *really* is.

10. Conclusions

The basic idea of the Law of Demeter is sound, no doubt. Positive experiences of applying it in practice are mentioned in [Lieb2] §13. Nevertheless, some details appear debatable, and the ramifications of the Law should at least be treated and explained more fully than has been done so far. This paper has mostly pointed out some questionable spots and discussed alternatives. The formulation of the Law in terms of classes is probably more useful than that in terms of objects, but the choice depends on the point of view. If we want to have a rule that can be checked at compile time the class form is certainly preferable; but with conventional untyped languages such checking is impossible in any case. Therefore, the Law appears to be more effective for *typed* languages.

The application of the Law of Demeter to C++ is promising, because the language is typed. We have seen, however, that the difference in paradigm between this procedural and only partially object-oriented language and the most archetypal object-oriented languages makes it non-trivial to translate programming rules from one environment into another. The original formulation of the Law for C++ clearly falls short of the mark. We have proposed a new formulation, which looks fairly logical and *is* compile-time checkable. More discussion and thinking about all details of the language will surely be needed before reaching a final result.

References

[Bobr] Daniel G. Bobrow *et al.*, CommonLoops: Merging Lisp and Object-Oriented Programming, *Proc. ACM OOPSLA '86 Conf. (Portland, Oregon, 1986),* ACM SIGPLAN Notices Vol. 21 No. 11, p. 17-29.

[Lieb1] Karl Lieberherr, Ian Holland, Gar-lin Lee, and Arthur J. Riel, An objective sense of style, *Computer (IEEE) Vol. 21 No. 6 (June 1988), p. 79-81 (The Open Channel).*

[Lieb2] K. Lieberherr, I. Holland, and A. Riel, Object-Oriented Programming: An Objective Sense of Style, *Proc. ACM OOPSLA '88 Conf. (San Diego, California, September 1988)*, to appear.

[Øste] Kasper Østerbye, Abstract Data Types with Shared Operations, *ACM SIGPLAN Notices Vol. 23 No. 6 (June 1988), p. 91-96.*

[Sakk] Markku Sakkinen, On the darker side of C++, *Proc. 2nd European Conf. on Object Oriented Programming (Oslo, August 1988),* Lecture Notes in Computer Science 322, Springer-Verlag 1988, p. 162-176.

[Stro1] Bjarne Stroustrup, The C++ Programming Language, Addison-Wesley 1986.

[Stro2] Bjarne Stroustrup, The Evolution of C++ : 1985 to 1987, *Proc. USENIX C++ Workshop (Santa Fe, New Mexico, November 1987).*

CHAPTER 4

DISCIPLINED INHERITANCE

First published in the Proceedings of **ECOOP '89** (Editor Stephen Cook), Third European Conference on Object Oriented Programming (Nottingham, July 1989), BCS Workshop Series, © Cambridge University Press 1989 (p. 39 - 56). — Reprinted with the permission of Cambridge University Press.

DISCIPLINED INHERITANCE

Abstract

Several other properties of objects are regarded as more important than inheritance. Inheritance is classified into two alternatives: incidental and essential. Incidental inheritance is a matter of implementation and software engineering, and should be completely encapsulated. Essential inheritance models important relationships in the problem domain, and should be completely visible. It is then attempted to model inheritance in terms of aggregation and dependence. A number of omissions and divergences and some surprising parallels are found in the treatment of inheritance in existing object-oriented languages and previous literature. This contribution does not pretend to close the case yet: for instance, intermediate forms between purely incidental and purely essential inheritance are not discussed.

Keywords: Inheritance, encapsulation, complex objects, type compatibility, delegation, programming discipline, class design.

1. Introduction

Many of us can agree with Bertrand Meyer when he states in the Preface of [Meye2], about the design of the Eiffel™ language:

> "The foremost influence has been that of Simula, which introduced most of the concepts twenty years ago, and had most of them right; Tony Hoare's remark about Algol 60 — that it was such an improvement over most of its successors — applies to Simula as well."

The first definition of SIMULA 67, later renamed simply Simula™, is [DaNyMy]. Single inheritance is well defined already there, although the word used is 'concatenation'. The language definition even in general has a down-to-earth style. Anthropomorphisms and overstatements have become part of object-oriented parlance later, apparently stemming from the Smalltalk (!) community. Examples of overstatement are 'active objects' and 'message passing' as applied to Smalltalk-80™ [GolRo] and similar languages. As [YokTo] explains, reconciling true asynchronous message passing with Smalltalk "message passing" is not quite simple.

'Inheritance' is an appealing word because its meaning in object-oriented programming (OOP) is so analogous to its usual meaning, which in turn is familiar to everybody. Still, probably because of this intuitiveness, there seems to be no common definition of 'inheritance'. Even in the ordinary meaning, there are many different kinds of inheritance, at least biological, juridical, and cultural; each of these has different rules.

To advance the state of OOP, we will need on one hand systems [AghHe, HaiNg, MinRo] that give the programmer more explicit choice over inheritance and other mechanisms that are fixed more or less ad hoc in conventional languages, and on the other hand tentative rules and restrictions that promote "better" programming [JohFo, LiHoRi, Sakk2, LieHo]. Here we stress the latter side of the coin, i.e. disciplined programming and software engineering. Features aiming at "exploratory programming" need not necessarily make the programmer into a Vasco da Gama or an Amundsen; she may well become Alice in Wonderland, never knowing what metamorphoses some seemingly innocent act may cause.

The main purpose of this paper is to suggest how the problems and ambiguities of inheritance could be controlled, and how inheritance

Eiffel is a trademark of Interactive Software Engineering Inc.

Simula is a trademark of Simula a/s.

Smalltalk-80 is a trademark of ParcPlace Systems.

could be modelled by other mechanisms. As I currently, agreeing with [Amer1,2], do not believe that inheritance is *the* central principle of OOP, it seems obligatory to present first the framework in which it will be discussed. In conscious provocation, I will list several properties of objects as more fundamental than either inheritance or *classes* — although I *do* believe in classes. We will also try to classify the intents of inheritance in a dichotomy.

The great variability of object-oriented terminology makes it problematic to write "cross-language" articles. Here we will not even try to speak about each specific language in the terms of its own literature, but neither will we succeed to establish a completely language-independent one-to-one relationship between terms and meanings. I apologise for the lack of concrete examples; they would have taken too much space.

2. What are objects

Here are again one person's views on what are the more important and less important characteristics of things to be called *objects*. Necessarily, most of these are requirements on the *system* (programming language or other) that manages objects.

The most important property of an object is **identity**. The identity must be unique within a given universe of objects and a permanent property of each object. The most degenerated possible kind of objects are those that have no other properties at all, but even they can be interesting — the whole general set theory is built on them.

The second property in importance is **integrity**. By object integrity I mean that no property of an object must be changed except by operations that *intend* and *have the right* to modify that object. The meaning of 'intend' here is very wide, covering even extremely indirect effects. — Weak support of object integrity in a language typically goes together with the lack of explicit identity: C++ [Stro1, Sakk1] is a good example.

The third property is that objects may in general be **created** and/or **deleted**. However, there can be objects that (at least conceptually) need not be created and/or cannot be deleted. The *possibility* of deletion can be a property of individual objects, and in that case need not be a permanent property.

The fourth property is that an object has a **type**. The type can be primitive; or it may be defined either *constructively* (structurally) from other types or *behaviourally* (as an abstract datatype), or in a mixed way.

In conventional OO systems the type of each object is permanent, but sometimes there are good reasons for letting even the type change [SkaZd].

One aspect of type is that an object may have **attributes** (data components). Attributes can be either non-object values or themselves objects. A special case of a non-object value is a *reference* (or surrogate), which is equal to the identity of some object or **nil**. In almost all OO languages and in some languages not considered object-oriented, e.g. CLU [Lisk&al], all *variables* are of a reference type. This has had consequences for the concept of inheritance (cf. §5). Some languages employ *capabilities* [AnPoWa], which combine references with access right information.

A second aspect of type is that an object may have **operations** (*methods*) that are executable. The difference between attributes and operations can become blurred in systems with triggers or access-based programming: what seems simply a read or update access to an attribute may actually cause the invocation of an operation.

A third aspect of type is that an object may have **processes** (*activities*). This is impossible in most current object-oriented languages: either they have only one sequential thread of control, or a process is a special kind of object or a separate implementation notion. In Simula, BETA [BKri&al], Actor languages, and newer concurrent/parallel OOPL's [YonTo], objects form also the units of concurrency (true or quasi-). In Emerald [Blac&al], operation invocations may proceed concurrently with each other and with their object's process. The Parallel Objects model of [CorLe] allows a high degree of controlled intra-object concurrency.

Sometimes [Viha, Sakk3] *abstract entities* such as (constant) integers are not considered to be objects. The argument is that it makes no sense to say that e.g. the integer 743 is "created" or "deleted", and saying that "the integer adds another integer to itself" feels hardly more appetising to a mathematically trained mind. Under this view, abstract entities need not have identities in the same sense as objects. — The opposite viewpoint *is* well motivated in [Lieb2] for the classless Actor languages. In a more typical language like Smalltalk-80, however, it appears hard to define consistently how a class that inherits e.g. from Integer and adds some instance variables should behave.

3. Less important properties of objects

Compared with Simula, most current object-oriented languages prohibit effective nesting, at least that of class definitions or their equivalents [BuhZa]. In compensation they enforce a stronger **encapsulation**, often so that only the operations and no attributes nor the process of an object can be directly accessed from outside the object itself. Although this is mainly beneficial, it sometimes requires operations concerning several objects to be defined in an unnaturally asymmetric manner; an exception is CLOS [Bobr&al] with its "multi-methods". Some languages allow more selective encapsulation, e.g. C++ with its **friend** declarations that can solve most of the asymmetry problem. — We will have more to say about encapsulation at several places.

One of the key concepts in OOP is **inheritance**. However, in spite of its flexibility and other attractions, unrestricted inheritance has been found to cause many problems as well. One fundamental contradiction is between inheriting specifications or behaviour on the one hand, and implementation or structure on the other hand. Also, single inheritance has lead to many unnaturally asymmetric constructions; this has later been remedied by multiple inheritance in many languages, but multiple inheritance has caused new problems (name conflicts etc.). We will base our constructions on multiple inheritance throughout. — Sometimes the concept of *delegation* [Lieb1] is proposed instead of or as a complement to inheritance [AghHe]. The claim that delegation is more powerful than inheritance was refuted by [Stei1], however. Note that some authors, e.g. [UngSm, HaiNg], use 'inheritance' approximately in the same meaning as some others use 'delegation'.

One feature connected to inheritance that is invariably counted as a principal factor in the flexibility of OOPL's is the redefinition of inherited operations. Nevertheless, this possibility is also a principal source of devious programming errors. We will suggest restrictions to such redefinitions in sections 6 and 7.

There can be **existential dependences** between objects; the attitudes of typical OO *programming* languages (garbage collection of totally unreachable objects) and most *database* systems (explicit, possibly cascaded deletions) are opposite to each other. There are some recent attempts [Kim&al, Sakk3] to combine these views. Existential dependences are a very special case of *constraints* in an object system. They can be efficiently enforceable, whereas more general systems of constraints very easily become undecidable. Dependences will be used in this paper as an auxiliary tool to model or even replace inheritance with aggregation. — The assertions of Eiffel seem to be an example of efficiently manageable constraints in an OOPL, but they are viewed mainly as a specification and debugging device. In ThingLab [Born] constraints are the essence of the system.

Most OOPL's are built around **classes**. A class is more than just a type: it is also a first-class object in e.g. Smalltalk and an "almost object" in e.g. C++ (a C++ class can have **static** variables that are accessible to all its instances). Typically, all operations are defined within classes and cannot be redefined in instance objects, although they are called "instance methods" in Smalltalk and many other languages. As an inheritance from C, C++ objects *can* have attributes that are pointers to functions, but these are not considered operations of the object. Inheritance also goes strictly between classes: an instance object cannot inherit anything but is instantiated with what the class has inherited.

There are languages that dispense with class as a primitive concept, e.g. Actors [AghHe] and Self [UngSm]. Instantiation is there substituted by replication of a *prototype* or *exemplar* object. This approach often goes together with replacing inheritance by delegation, but not always [LaThPu]. Our presentation will mostly be independent of the class-prototype controversy; I am suggesting elsewhere [Sakk4] that to reject classes is to throw the child out with the wash.

In Algol 68, Ada®, and some other languages, although *types* are only compile-time entities, formal type parameters can be used to define **generic** (or *parametric*) packages, procedures, new types, or whatever (depending on the language). The advantages of genericity are thoroughly discussed in [Meye1,2]. Generic classes have not been implemented in many OOPL's besides Trellis/Owl™ [Scha&al], Mode [Viha], and Eiffel. Of course, the whole principle is relevant only to strongly typed class-based languages; the majority of currently popular languages is weakly typed.

It is not common to regard the type of an object (or of a non-object value) as a first-class value that can e.g. be assigned to a variable of type 'type'. However, this approach has been chosen in the Mode language, and it looks like a sensible enrichment of the type system. It does *not* mean that the type of an object could be magically changed by updating its "type field".

Ada at least *was* a registered trademark of the United States Government (AJPO).

Trellis is a trademark of Digital Equipment Corporation.

4. Essential vs. incidental inheritance

There is often *too much* inheritance in object-oriented programming. Programmers can sometimes apply inheritance when plain aggregation would be more suitable, but even some languages as such force too much of a good thing. The Smalltalk tradition that there must be a universal superclass (Object) causes problems, especially if it is still enforced with multiple inheritance. This requirement does not exist in Simula and its direct followers.

There are also many *flavours* of inheritance. The difference between the views of inheriting specification (behaviour) or implementation was mentioned several times at ECOOP'88 in Oslo (panel discussions are not recorded in [GjeNy]), often saying the former to be typically European and the latter typically American. We propose two new terms that seem to clarify the picture a little at least for the present purposes: **essential** and **incidental** inheritance. Inheritance of implementation only is always incidental. Inheritance of specification is essential, whether implementation is inherited also or not. In the classification of [HaiNg], the case "Code Sharing and Reuse" corresponds to our incidental inheritance, the others to different facets of essential inheritance: "Type Theory Inheritance" implies inheritance of implementation as well, while "External Interface Inheritance" and "Simple Polymorphism" do not. Ian Holland has proposed the following interpretation:

> Incidental inheritance seems to appear as a result of software engineering and program design. Essential inheritance occurs as a result of domain analysis and system design.

This captures the heart of the matter.

We do not claim the distinction between essential and incidental to be absolutely sharp. One might say that as the commonality decreases and the differences increase between two "like" types [WegZd], their relationship becomes more and more incidental. A simple example that could be regarded as incidental inheritance of specification can be drawn from the ever-popular domain of data structures. A stack and a queue can have the same set of operations, and even with the same signatures. Nevertheless, if a software designer should change the operations of a stack, this need not necessarily affect the interface of a queue. There might further be e.g. a 'button' class in the same system, which could also have a 'push' operation like a stack. The connexion between these operations would be a purely incidental name collision — we have not got enough space here to explain how we would completely exclude cases like this from "inheritance" (cf. §9).

Examples of insightful recent papers on essential inheritance are [WegZd] and [LKnu]. On the contrary, the interesting paper [Snyd3] (a slightly modified version of [Snyd2] with a CommonObjects example added) focusses on incidental inheritance. Snyder explicitly uses 'inheritance' in the meaning 'inheritance of implementation' (so does [JohFo]). He does mention inheritance of specification, calling it 'subtyping'. The same terminology is used in [Amer1], a paper that is fully relevant to non-parallel languages as well. Readers should note the difference in terms between these papers and ours. However, we will actually not study situations in which *only* specification is inherited, because there would not be much to say about them (well, [Amer2] says a lot). We will thus stay on the common ground that practically all authors call inheritance.

Since we consider the properties of §2 more important than inheritance, we will not accept such features of inheritance that would clearly be in conflict with any of those properties. As pointed out in [Amer1], inheritance will probably play a less important role in parallel than sequential languages. We will not even discuss inheritance of processes, although already Simula offers the **inner** keyword for that purpose. It allows the superclass to define at what point the body (process) of any subclass shall be executed: a kind of converse to the **super** construct of many languages that allows a subclass operation to define that a superclass operation is to be invoked.

5. Inheritance as aggregation

Contrary to almost all non-OO languages, almost all OO languages enforce the principle of totally indirect *aggregation* (composition): variables in objects cannot contain subobjects directly but only references to other objects. The obvious advantage is that classes remain really independent of each other's implementations. Certainly the best-known exception that treats aggregation (both arrays and records) like "ordinary" programming languages is C++. EXTRA [CaDeVa] is similar, but more in Pascal style and handles even sets. Both C++ and EXTRA *allow* a class designer to use pointers (references) whenever more appropriate.

The principle of indirect aggregation must be a major culprit for the overuse of inheritance in OOP. It is also the main reason why we need a concept of incidental inheritance at all, instead of mere aggregation: the parent part normally *is* physically concatenated to the non-inherited part of an object, conforming to the original Simula approach (cf. §1). However, if we add three restriction capabilities that are missing from most languages today, we can model inheritance (both kinds) by aggregation. Aggregation is a much less ambiguous concept than inheritance. We will concentrate on *data* (instance variables) and not speak anything yet about inheriting *operations*. It is actually the more important half of the question, but it depends on the kind of inheritance, and will be discussed in the following sections. — We will speak in terms of classes, but the discussion is applicable to prototype-based languages as well.

The "mathematical difference" of a subclass object and a corresponding superclass object is not defined in the conventional OO view, or in any case it is not an object. This can be seen as a defect in the object or class algebra; the problem has been recognised in [Stei2]. We strive to a model in which the difference will always be an object. We will thus allow even objects that have a "negative compound type", while [Stei2] does not permit such types to be instantiated; but these objects cannot exist alone, only as parts of complex objects. In [WegZd] §7 it is noted that it is often most convenient to define largely similar classes by giving only their incremental differences. This thought is not far from Stein's type expressions.

To be definite, suppose class C has the immediate superclasses D_1, ..., D_n. Let us regard any instance O of class C as a *complex object*, consisting of a full-fledged object P_i of each class D_i, and one further object O^- that contains the non-inherited instance variables (components) of O. Let us call the class of O^- analogously C^-. With a slight change to common OOPL usage, we could actually define C^- first without naming the superclasses, and then C as a "sum class". This would then explicitly be "programming by difference". The exact structure of the complex object O turns out to be different in essential and incidental inheritance, and so is deferred to the following sections.

To obtain the usual semantics of inheritance, the three restrictions we must be able to put on every subobject (parent component) P_i of O are as follows:

(1) P_i is deleted when and only when O has been deleted. (In the terms of [Sakk3], P_i is "immediately completely dependent" on O.)

(2) The connexion between O and P_i is permanent.

(3) O must export no reference (surrogate) to P_i neither to its other components nor to its clients (objects that invoke O's operations).

In fact, these restrictions apply to O^- as well; we can denote that subobject also by 'P_0', and C^- by 'D_0'. The inheritance model of conventional OOPL's certainly fulfills these conditions: (1) and (2) because each P_i is physically contained in O, (3) because the components are not objects at all and thus cannot be referred to.

The class C^-, if it really needs a *super*class to be meaningful, is a kind of mirror image of an *abstract class*, which needs a *sub*class to be meaningful. Our present approach implies that even abstract classes can be instantiated, but their instances must always be components of other, subclass objects. We can say that they, too, have negative compound type. Thinking in terms of difference classes seems to make the relationship between subclass and superclass more symmetric. A further advantage of difference classes is that some of them may be sensibly combined with different parent classes, thus reducing duplicated declarations.

The "mix-in" classes (flavors) of Flavors (mentioned in [Snyd3] but not in [Moon]) seem to be largely equivalent to our difference classes. In Flavors, parents are called 'components' as they are here, but the word has a slightly different meaning — instance variables are *not* components. Further, the "mixing" principle means that there will generally be no real subobjects in a Flavors object, because components lose their integrity. In most other OOPL's a parent object essentially exists within each child object, but one cannot refer to it as such (as noted above): it has integrity but no identity.

6. Modelling incidental inheritance

Now we take up the questions of inherited operations and complex object structure. One would expect it to be more straightforward to model or replace incidental than essential inheritance by aggregation. This proves to be the case. Suppose that class C has the parent classes D_1, ..., D_n, O is an instance of C, and P_i (i = 1, ..., n) are the corresponding instances of the parent classes, like in the previous section. Incidental inheritance means that we do not want to say that C *is-a* D_i, not even that C *like* D_i [WegZd]. Therefore, a client of O will not expect to get from it any of the services provided by D_i objects. We certainly intend that C *is-a* C^-, so we must not count $C^- = D_0$ as a parent class in this section!

I found a striking parallelism between [Snyd2,3] and [Lieb&al, LiHoRi, Sakk2] when parent classes in the former papers are equated with component classes in the latter ones. As regards *attributes* (instance variables), Snyder's requirement that instance variables of a parent class should not be directly accessible in a child class is equivalent with the rule in the Law of Demeter™ that a class's methods should see only the immediate structure (components) of the class. (Snyder's requirement as such is also presented in [LiHoRi] as an additional rule that makes the difference between the *strong* and the *weak* Law of Demeter.) As regards *operations* (instance methods), Snyder's requirement that "a class may refer to non-immediate ancestors only if they are exposed via the intervening classes" is again equivalent with a rule in the Law of Demeter: operations of the components of a class must not be directly invoked from outside the class.

In the implementation of CommonObjects [Snyd1], our complex object O would be a tree of subobjects such that O^- is its root and $P_1, ..., P_n$ are its leaves. This is the most logical representation; we will just regard the subobjects as first-class objects too.

Neither multiple inheritance nor even repeated inheritance of the same parent [Meye2] presents many additional problems in this approach. In the terms of [LKnu], only casual horizontal name collisions can occur, and these can be resolved by qualification. Thus at worst, O must request any otherwise ambiguous inherited operation from an explicitly specified parent object. Both Snyder and Demeter require that the *attributes* of a parent component P_i not be directly available to (the operations of) O, and the *operations* of P_i not be directly available to other components nor clients of O. This does not prevent O from exporting any or all of P_i's operations to O's clients, renaming them or not, but the clients will see them as O's own operations.

Our approach actually comes close to *delegation* here (cf. [Snyd3] §2.3) in that we model the inherited parts as objects in their own right. We make the significant difference to conventional delegation that no request within a P_i for any of its own operations shall be dynamically changed into a request for an operation of O (O^-) with the same name, because we do not regard it as the "same" operation. This, I think, closes the last leak that still remained in [Snyd3] in encapsulating inheritance. Late binding of operations, which many languages enforce in all situations, can make a parent class D_i in a way a client of the child class C: encapsulation is severely violated. Weird situations are possible even in a generally well-disciplined language like Eiffel ([Meye2] §11.2): If an operation *f* of class D_i is redefined in class C, the original *f* can be renamed and invoked within C by the new name. But there is no way whatsoever for other operations of D_i to invoke the original *f*, guaranteedly uninterfered by any later subclass definition.

Demeter is a trademark of Northeastern University.

Many languages, including CommonObjects, do offer a means to specify that some particular operation invocation is to be bound early (to the class of the *invoking* operation). Some languages, at least Simula and C++, additionally allow an ancestor class to declare which of its operations can be redefined in descendant classes (are **virtual**) and which not. BETA further allows a descendant class to modify an inherited virtual function to non-virtual for its own descendants, which is a very logical possibility. Here we have a situation different from all these: early binding of D_i's operations is a property of C, not of D_i itself, and is a consequence of the inheritance from D_i to C being strictly incidental and therefore encapsulated.

A property distinct from (but not quite independent of) the redefinability of operations is their *visibility*. One language that gives a good control of operation (and attribute) visibility is Trellis/Owl. Likewise, current C++ [Stro2] has the alternatives: **private**, visible only within the class itself; **protected**, visible to subclasses also; **public**, visible everywhere in the programme. In the special case of incidental inheritance, however, we suggest that even protected operations of D_i should *not* be invocable in C — since C is not a *subtype* of D_i. C++ originally had only the alternatives *private* and *public*. Some others, including Eiffel, have only *protected* and *public*, which I consider a worse selection. Unfortunately all operations are public in some OOPL's, including Smalltalk.

Incidental inheritance provokes the question about the visibility of *classes* [BKri&al, BuhZa]. The Law of Demeter also restricts the accessibility of classes to each other, but rather with "need to know" principles, not on the basis of nesting. If a class D_i is designed only to support C and perhaps some further class E, then the very existence of D_i could well be hidden from the clients of C and E.

7. Modelling essential inheritance

After the completely incidental inheritance in the previous section, we will now examine the other extreme. This is essential inheritance with the very strong requirement of *complete compatibility* [WegZd]:

> A subtype is completely compatible with its supertype if it has the same domain as the supertype and, for all operations of the supertype, corresponding arguments yield corresponding results.

("The same domain" here does not preclude the subtype from possessing additional attributes and operations besides the inherited ones.) As shown in [WegZd], this implies the *principle of substitutability*:

An instance of a subtype can always be used in any context in which an instance of a supertype was expected.

Because we have wanted to encapsulate attributes, we could omit the "same domain" requirement from the first definition; this obviously would not disturb the principle of substitutability. If we were more interested in specifications than implementations, the definition could even better be expressed in terms of *traces* [McLe].

Let C be a subclass of D_i, as in the previous sections, but now with complete compatibility (CC). That requires every public operation *f* of D_i to be available also in C. Let again O be an instance of C and P_i the corresponding instance of D_i. If any call of O.*f* is just directed to the D_i component as a call to P_i.*f* with the same argument list, we trivially obtain CC. Are there any simple rules that could ensure CC in non-trivial situations? We propose the following sufficient but not necessary condition:

(1) The subclass operation O.*f* must invoke the superclass operation P_i.*f* and return the result given by P_i.*f*. Other actions of O.*f* must not modify P_i.

If there are *subclass-visible* (protected) operations, we need an additional rule (again sufficient but not necessary):

(2) No subclass operation must directly invoke any *modifying* protected operation of the superclass.

This is because such protected operations might leave the superclass object in a state that could not result from applying only public operations. It is also logical to require that all protected operations of the superclass remain protected operations of the subclass. We may allow them to be redefined subject to condition (1); condition (2) then implies that this is only possible with non-modifying protected operations.

If some subclass operations do not obey the rules (1) and (2), it might still be possible to prove complete compatibility e.g. along the lines of [Lamb], using traces. A method that employs traditional pre- and postconditions together with abstraction functions to prove behavioural compatibility is presented in [Amer2].

The complex object structure required by essential inheritance can be a little different from the pure tree structure suggested in the previous section. In an extreme case, the root object O^- and thus also the class C^- can be omitted entirely: if O is merely a "sum" of P_1, ..., P_n and no two distinct ancestor objects have any common public or protected operation. (The next section will present situations in which *one* ancestor object can be reached by different inheritance paths.) This means that we can add "orthogonal" classes to existing classes in essential inheritance without adding levels to subobject trees, i.e. without introducing more levels of abstraction. Normally, the subclass C redefines and adds some operations, of course; a tree structure *is* then needed. However, all other operations of the parent classes remain as visible in the context of C as they originally were.

We have seen that complete compatibility requires the opposite on the visibility of inherited operations, when compared to incidental inheritance. (Obviously, we must not prohibit late binding of superclass operations, as we did in the previous section: all operations must be searched from the complex object level.) In our aggregation model we seem to get a conflict with the Law of Demeter by looking more than one level deep into the subobject tree. We can resolve the conflict by specifically allowing this exception to the law; but the need to make an exception should make us cautious about essential (visible) inheritance.

In typical cases of essential inheritance, the CC property cannot be required. This means that public operations of a descendant class may return different results than those of an ancestor class, and not all public operations of an ancestor are even offered by a descendant. This again implies that run-time checks may be necessary even in strongly typed languages. An important case is *read-only substitutability*, defined in [WegZd] as follows:

> An instance of a subtype can always be used in read-only mode in any context in which an instance of a supertype was expected.

The difference to CC is that updating operations of an ancestor may work incompatibly with a descendant (as long as they preserve any invariants that the ancestor class may have) or need not be applicable at all: the extreme case is a constant object. 'Read-only' can be interpreted as 'having benign side effects' [Lamb].

Repeated essential inheritance of a parent class is not straightforward. Although I do not know of any other language than Eiffel allowing direct repeated inheritance, it can easily arise indirectly in any language that permits multiple inheritance (cf. next section). If C inherits D *m* times, we cannot say that C *is-a* D; instead, C *is-a set* of D, although a set with a fixed number of elements. Handling such a situation in an orderly way would require intrinsic set operations, which most OOPL's have not got. Practically all languages impose some conditions, e.g. on explicit renaming or implicit operation lookup, that effectively leave at most one path of inheritance essential in our sense.

8. Fork-join inheritance

Explaining essential inheritance by aggregation gets tough when a class inherits a non-immediate ancestor over more than one path. For lack of a generally agreed term, we will call this situation *'fork-join inheritance'*. The most simple example possible is sufficient to show the difficulties: let D and E be parents of C, and F a parent of both D and E. We will look at how it is handled in Eiffel ([Meye2] §11.6.2), in the coming (?) version of C++ with multiple inheritance ([Stro2] §3), and in the theoretical paper [LKnu]. The last paper does not discuss operations, which are for our purposes both more important and more difficult than attributes.

Note that we get no new kind of problem if one or more of the four inheritance links involved is totally incidental: C cannot then "see" F over two paths. Otherwise, if the situation we want to model is such that the D and E subobjects of C should each have its own private F subobject, we have repeated inheritance (see end of previous section), just indirectly. The new, interesting case arises when we require a shared F subobject, thus a fork-join relationship on the instance level, too.

Stroustrup's model allows every class to declare any or all of its parents **virtual**. If F is declared virtual in both D and E, it means that the D and E parts of a C object share a common F part; otherwise each has its own. (In the example of [Stro2], C declares F also as a (direct) **virtual** parent, but this can hardly be necessary in general.) The approach is clean and understandable; most importantly, it preserves the integrity of the F subobject. It fits the model of §5 perfectly. Nevertheless, as discussed in [Stro2], programming the operations in fork-join inheritance is somewhat tricky: each class will typically need, for each public operation, an accompanying *protected* operation (with a different name) that does only "the own stuff" of the particular class. The public operations of each class must then explicitly call the appropriate protected operations of all ancestor classes. — Built-in modes of operation combination in Flavors [Moon] and CLOS [Bobr&al] probably eliminate such programming chores in most situations.

Meyer presents a "transcontinental drivers" example that cannot be simply and naturally reduced into object aggregation. Part of the attributes inherited from F by D and E are shared, part replicated. Those that are replicated are renamed in C so that there are no name conflicts: the repeated inheritance rule of Eiffel says that exactly those inherited attributes and operations shall be shared that have *not* been renamed along any of the inheritance paths. The problem is that the F part gets effectively split into two. The integrity of subobjects is thus violated, somewhat like in Flavors (§5).

Lindskov Knudsen suggests the responsibility for sharing or replicating to be divided between C and F: just the opposite of the C++ approach. Every (non-inherited) attribute of every class must be declared either *singular* or *plural*; singular attributes will always be shared in a fork-join inheritance. Every class must declare whether its inheritance method is *unification* or *intersection*. If it is unification, then any plural attribute inherited along several paths from a common ancestor is replicated. If it is intersection, then even such plural attributes are shared. Clearly, the integrity of subobjects can get violated in this approach, too, if we try to reduce inheritance to aggregation.

I claim that these latter two inheritance models allow anomalies that make them not generally recommendable. Obviously, the approach of [Meye2] is strictly more general (or less disciplined) than that of [LKnu]. The same kind of anomaly still creeps up in both models. The really fatal defect is that any operation of F, D, or E, public, protected, or private, that both updates shared attributes and accesses replicated attributes, may cause unwanted side effects between the D and E parts of a C object. All operations of F, D, and E must therefore be checked, and the code-sharing advantage of inheritance is lost.

Another argument is that the examples given to illustrate these two inheritance models look like patches to bad class design in the ancestor class. A relational database expert would probably identify the heart of the problem as F not being in second normal form. We look at the example of [Meye2], which is smaller. The "root" class (F) is called Driver: it contains attributes such as Age and Number-of-violations. The intermediate classes are called French-driver and US-driver: both inherit all attributes of Driver and have some of their own. The lowermost class is called French-US-driver: it inherits all attributes of both French-driver and US-driver. Some Driver attributes are shared, e.g. Age, some are renamed and thus replicated, e.g. Number-of-French-violations, Number-of-US-violations.

We note that French-US-driver is not a *subtype* of any of its ancestor classes, because it has not all their attributes. It would be a subtype of two of them if all replicated Driver attributes were renamed only on one path; but then French-driver and US-driver would not be in an equal position. The whole class seems to have very little purpose other than to coerce the splitting of Driver. The approach begins to look futile indeed when we consider that every combination of two or more countries that is needed to model some multinational driver in the system requires a similar class of its own.

I propose that the Driver class should be explicitly divided into two classes in the first place: Person (containing Age and the other attributes that are now shared) and Only-driver (= Driver ◘ Person, containing Number-of-violations and the other replicated attributes), such that Person is a shared (virtual) parent of Only-driver. French-driver and US-driver would inherit (without sharing) from Only-driver. In fact, it could be better not to regard the relationship of Person and Only-driver as inheritance at all, but merely to declare an attribute of type '[reference to] Person' in Only-driver. That way, we need not necessarily delete a person from the system just because he loses his driving licence. — In the model of [LKnu], every class that has both singular and plural attributes should similarly be split into two classes. Both unification and intersection can then be handled within our model.

9. Topics to be pursued

There are some aspects connected to the theme of this contribution that I already have ideas about or that should be taken into account before the model of inheritance as aggregation is completed. Some thoughts surfaced too late to be developed, when finishing this paper for publication. Besides, the paper is too long already, so a short listing shall suffice.

In multiple inheritance, some ancestors of a class may be incidental and some others essential. The discussion of sections 6 and 7 is easily applicable to this case. A problem that we have not treated is a partly incidental, partly essential inheritance relationship between a subclass and *one* parent class: for instance, only part of the parent's public operations are public in the child as well. We have not spoken much even about essential inheritance weaker than completely compatible. Errors and misunderstandings in programming may most easily happen in this "gray area". I feel that it could perhaps be possible to "normalise" classes into difference classes (very analogously to database normalisation) such that inheritance could be factored into purely incidental relationships and CC essential relationships. (Cf. "consistent subsets" and "inheritance packages" in [Amer1] §3.3.)

In contrast to the above, the boundary between incidental inheritance and ordinary aggregation does not look problematic at all. Any or all of the restrictions (1-3) in §5 can be relaxed or removed without causing obvious inconsistency. For instance, the composite objects of ORION [Kim&al] need not obey rules (2) and (3) nor the "only when" part of rule (1).

In my opinion, any self-respecting framework of multiple inheritance must offer an elegant and consistent treatment of "multi-methods" [Bobr&al, Øste]. This can be difficult, e.g. because the "specificity" order of class combinations is only partial.

It seems that the current approach can solve "the *self* problem" as presented in [Lieb1] §6. I think it can be worthwhile for some purposes to introduce at least two further reflexive pseudo-variables ("reflexive pronouns"), both usually denoting a smaller complex object than *self*, but mostly larger than *super*. The first one would denote the subobject to which the current operation belongs (*my-node* in CommonObjects [Snyd1]). The other one would denote the subobject whose operation has been originally called from outside the complex object; it would be interesting only if restriction (3) of §5 is omitted.

The importance of *names* (identifiers invented by the programmer) in identifying operations (also attributes and classes) can and should be diminished. We hinted at this in §4; many name clashes are simply false friends that can be thus avoided. The use of *titles* (roles) instead looks promising; titles can also be employed to create objects that are an intermediate of class and instance [Sakk4].

Acknowledgements

This work has been supported by the Academy of Finland (initially) and the Ministry of Education (doctoral programme in information technology). Correspondence with the Demeter group of Northeastern University (Karl Lieberherr, Ian Holland, and others) has partially inspired me to this particular line of research.

I am much indebted to Ian Holland for a thorough commentary on the first version of this paper; it has caused many revisions. The conference reviewers' comments were useful, too. I also thank Alan Snyder (Hewlett-Packard Laboratories) for clarifying some points, and Pierre America (Philips Laboratories) for providing some relevant papers.

References

[AghHe] Gul Agha and Carl Hewitt, *Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming*, [ShrWe] 49-74.

[Amer1] Pierre America, *Inheritance and subtyping in a parallel object-oriented language*, [Bézi&al] 234-242.

[Amer2] Pierre America, *A Behavioural Approach to Subtyping in Object-Oriented Programming Languages*, Philips Research Laboratories, Eindhoven (The Netherlands) 1989.

[AnPoWa] M. S. Anderson, R. D. Pose, and C. S. Wallace, *A Password Capability System*, Comp. J. 29:1 (1986) 1-8.

[Bézi&al] Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman (Ed.), *ECOOP '87 European Conference on Object Oriented Programming (Paris, June 1987) Proceedings*, Lecture Notes in Computer Science 276, Springer-Verlag 1987.

[Blac&al] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy, *Object Structure in the Emerald System*, [Meyr1] 78-86.

[Bobr&al] Daniel G. Bobrow et al., *Common Lisp Object System Specification*, ACM SIGPLAN Notices 23: special issue (September 1988).

[Born] Alan Borning, *The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory*, ACM ToPLaS 3:4 (October 1981) 353-387.

[BKri&al] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard, *The BETA Programming Language*, [ShrWe] 7-48.

[BuhZa] P. A. Buhr and C. R. Zarnke, *Nesting in an Object-Oriented Language is NOT for the Birds*, [GjeNy] 128-145.

[CaDeWa] Michael J. Carey, David J. DeWitt, and Scott L. Vandenberg, *A Data Model and Query Language for EXODUS*, [SI88] 413-423.

[ChiD'A] E. Chiricozzi and A. D'Amico (Ed.), *International Conference on Parallel Processing and Applications (L'Aquila, Italy, September 1987) Proceedings*, North-Holland 1988.

[CorLe] Antonio Corradi and Letizia Leonardi, *How to embed concurrency within an object environment: Parallel Objects*, [ChiD'A] 79-84.

[DaMyNy] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard, *SIMULA 67 Common Base Language*, Norwegian Computing Center 1968 (No. S-2).

[GjeNy] S. Gjessing and K. Nygaard (Ed.), *ECOOP '88 European Conference on Object Oriented Programming (Oslo, August 1988) Proceedings*, Lecture Notes in Computer Science 322, Springer-Verlag 1988.

[GolRo] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley 1983.

[HaiNg] Brent Hailpern and Van Nguyen, *A Model for Object-Based Inheritance*, [ShrWe] 147-164.

[JohFo] Ralph E. Johnson and Brian Foote, *Designing Reusable Classes*, Journal of Object-Oriented Programming 1:2 (June/July 1988) 22-30,35.

[Kim&al] Won Kim et al., *Composite Object Support in an Object-Oriented Database System*, [Meyr2] 118-125.

[LaThPu] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh, *An Exemplar Based Smalltalk*, [Meyr1] 322-330.

[Lamb] David Alex Lamb, *Benign Side Effects*, Inf. Proc. Lett. 29:6 (December 1988) 301-305.

[Lieb&al] Karl Lieberherr, Ian Holland, Gar-lin Lee, and Arthur J. Riel, *An objective sense of style*, Computer (IEEE) 21:6 (June 1988) 79-81 (The Open Channel).

[LiHoRi] K. Lieberherr, I. Holland, and A. Riel, *Object-Oriented Programming: An Objective Sense of Style*, [Meyr3] 323-334.

[LieHo] Karl J. Lieberherr and Ian Holland, *Formulations and Benefits of the Law of Demeter*, submitted paper (1988).

[Lieb1] Henry Lieberman, *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, [Meyr1] 214-223.

[Lieb2] Henry Lieberman, *Concurrent Object-Oriented Programming in Act 1*, [YonTo] 9-36.

[LKnu] Jørgen Lindskov Knudsen, *Name Collision in Multiple Classification Hierarchies*, [GjeNy] 93-109.

[Lisk&al] Barbara Liskov et al., *CLU Reference Manual*, Lecture Notes in Computer Science 114, Springer-Verlag 1981.

[McLe] John McLean, *A Formal Method for the Abstract Specification of Software*, JACM 31:3 (July 1984) 600-627.

[Meye1] Bertrand Meyer, *Genericity versus Inheritance*, [Meyr1] 391-405.

[Meye2] Bertrand Meyer, *Object-oriented Software Construction*, Prentice-Hall 1988.

[Meyr1] Norman Meyrowitz (Ed.), *OOPSLA '86 Conference Proceedings (Portland, Oregon, 1986)*, ACM SIGPLAN Notices 21:11 (November 1986).

[Meyr2] Norman Meyrowitz (Ed.), *OOPSLA '87 Conference Proceedings (Orlando, Florida, 1987)*, ACM SIGPLAN Notices 22:12 (December 1987).

[Meyr3] Norman Meyrowitz (Ed.), *OOPSLA '88 Conference Proceedings (San Diego, California, 1988)*, ACM SIGPLAN Notices 23:11 (November 1988).

[MinRo] Naftaly H. Minsky and David Rozenshtein, *A Law-Based Approach to Object-Oriented Programming*, [Meyr2] 482-493.

[Moon] David A. Moon, *Object-Oriented Programming with Flavors,* [Meyr1] 1-8.

[Øste] Kasper Østerbye, *Abstract Data Types with Shared Operations,* ACM SIGPLAN Notices 23:6 (June 1988) 91-96.

[Sakk1] Markku Sakkinen, *On the darker side of* C++, [GjeNy] 162-176.

[Sakk2] Markku Sakkinen, *Comments on "the Law of Demeter" and C++,* ACM SIGPLAN Notices 23:12 (December 1988) 38-44.

[Sakk3] Markku Sakkinen, *Objects, non-objects, and existential dependences*, submitted paper (1988).

[Sakk4] Markku Sakkinen, *Between classes and instances, aided by titles,* submitted paper (1989).

[Scha&al] Craig Schaffert et al., *An Introduction to Trellis/Owl*, [Meyr1] 9-16.

[ShrWe] Bruce Shriver and Peter Wegner (Ed.), *Research Directions in Object-Oriented Programming*, The MIT Press 1987.

[SI88] *SIGMOD '88 Conference (Chicago, June 1988) Proceedings*, ACM SIGMOD Record 17:3 (September 1988).

[SkaZd] Andrea H. Skarra and Stanley B. Zdonik, *Type Evolution in an Object-Oriented Database*, [ShrWe] 393-415.

[Snyd1] Alan Snyder, *CommonObjects: An Overview*, ACM SIGPLAN Notices 21:10 (October 1986) 19-28.

[Snyd2] Alan Snyder, *Encapsulation and Inheritance in Object-Oriented Programming Languages*, [Meyr1] 38-45.

[Snyd3] Alan Snyder, *Inheritance and the Development of Encapsulated Software Systems*, [ShrWe] 165-188.

[Stei1] Lynn Andrea Stein, *Delegation Is Inheritance*, [Meyr2] 138-146.

[Stei2] Lynn Andrea Stein, *Compound Type Expressions: Flexible Types in Object Oriented Programming*, [Meyr3] 360-361.

[Stro1] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley 1986.

[Stro2] Bjarne Stroustrup, *The Evolution of C++ : 1985 to 1987,* USENIX C++ Workshop (Santa Fe, New Mexico, November 1987) Proceedings.

[UngSm] David Ungar and Randall B. Smith, *Self: The Power of Simplicity*, [Meyr2] 227-242.

[Viha] Juha Vihavainen, *The Programming Language Mode Language Definition and User Guide*, Report C-1987-50, University of Helsinki (Finland), Department of Computer Science 1987.

[WegZd] Peter Wegner and Stanley B. Zdonik, *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like,* [GjeNy] 55-77.

[YokTo] Yasuhiko Yokote and Mario Tokoro, *Concurrent Programming in ConcurrentSmalltalk*, [YonTo] 129-158.

[YonTo] Akinori Yonezawa and Mario Tokoro (Ed.), *Object-Oriented Concurrent Programming*, The MIT Press 1987.

CHAPTER 5

A CRITIQUE OF THE INHERITANCE PRINCIPLES OF C++

First published (except Corrigendum) in Computing Systems Vol. 5 No. 1 (Winter 1992), p. 69 - 110. © University of California Press and USENIX Association 1992. — Reprinted with the permission of the Regents of the University of California. The journal version contains a few small editorial changes not found in this chapter. **Corrigendum** to appear in Computing Systems; published in parallel with permission.

A CRITIQUE OF THE INHERITANCE PRINCIPLES OF C++

Abstract

Although multiple inheritance (MI) is already a feature of the C++ language, there is a debate going on about its good and bad sides. In this paper, I am defending MI. At the same time, many of the rules and principles of inheritance in current C++ seem to me to need improvements, The suggested modifications are relatively simple, do not introduce many new reserved words, and should not affect other parts of the language.

I do not find the current rules totally adequate even for single inheritance (SI). The problems lie in the meaning of access levels and in the redefinability of virtual functions. Additional inconsistencies in C++ virtual functions appear in so-called independent multiple inheritance (IMI), which is in principle the easy case of MI. The most difficult problems are caused by so-called fork-join inheritance (FJI), which is the most complicated kind of inheritance.

One essential cause of complexity is *private* inheritance because of its intransitive nature. The main idea suggested here is that, simply put, private inheritance should be implicitly "non-virtual" and public inheritance "virtual". This is actually a simplification of the language, at least from the programmer's if not from the implementor's point of view. The main rule has also been generalised to arbitrary combinations of private and public inheritance, with some restrictions on legal combinations. I

sincerely think that the current C++ rules will be very harmful if programmers start developing complex class hierarchies in which FJI is applied. On the opposite, the new rules suggested here should behave consistently even in complex situations — but demonstration is so far missing, of course. The same principles should be applicable to other object-oriented languages beside C++.

MI increases the complexity of the language in any case; Cargill [1991a] has therefore required good examples of its advantages to make it worthwhile. I think that Waldo's [1991a] example is convincing enough for IMI, but I join in the quest for equally good examples using FJI.

1. Introduction

This article is aimed at an audience that has some previous understanding of both object-oriented programming (OOP) and the C++ language (some parts will require even a quite detailed knowledge of C++). A reader who thinks that the two are almost synonyms, or that OOP and Smalltalk-80™ are almost synonyms, should probably first read one of the good tutorials and books on OOP that are available today.

Be warned first that the author is a convinced opponent of C and all C-based languages, for general-purpose programming [Sakkinen 1988, 1991]. In spite of that, there will be several constructive suggestions for improvements to C++ in the article (of course I would prefer the fundamental ideas to be adopted into inherently better languages). Secondly, I am currently a rather pure theoretician, not having really programmed in *any* language for quite some time. I had some experience of C++ prior to Release 2.0, and while finishing the paper, have been able to test some questionable things on a Release 2.1 compiler (Hewlett-Packard). Thirdly, try to bear with my British spelling if it has not been Americanised by the editorial staff.

The initial motivation for this article was to examine *multiple* inheritance (MI). There is still a debate on whether MI is at all necessary in object-oriented systems, and whether its true advantages outweigh the complexity that it introduces into a language, particularly C++. I do believe in MI, especially from a theoretical standpoint. It was therefore surprising to me that among the 30 or so participants of the ECOOP'91[1] workshop on "Types, inheritance and assignments", a majority answered "No" when asked if they would include MI in *their* next object-oriented language. Perhaps the answer was motivated mainly by implementation considerations — I forgot to ask about that.

In C++, multiple inheritance has already been defined and implemented; although every independent new implementer must suffer the tedium of MI. As there is no official standard for the language yet, Tom Cargill has been campaigning in [1991a] and elsewhere at least for a moratorium against MI. He has recently been countered by Jim Waldo [1991a]. Here I will try to give some further arguments in favour of MI.

Unless we agree with the verdict that multiple inheritance should be banished from C++, the next important question is *how* it should be defined. Is the current approach adequate? Originally I really liked it [Sakkinen 1989]. However, there have appeared some problems [Baclawski 1990; Snyder 1991]. I will try to elaborate on these problems in the larger part of this paper. They proved to be more numerous and more complex than I had thought at the start of writing. Another surprise was that some problems pertain already to single inheritance (SI).

The plot of the rest of this article is as follows. I will try to summarise shortly the most relevant points of Cargill and Waldo on MI in Section 2, with some references to other literature. Section 6 draws some conclusions, finishing in optimism on the feasibility of multiple inheritance. At the end we have the obligatory acknowledgements and reference list. The beef of the hamburger is in the middle.

Section 3 treats issues that are relevant even to SI, especially ideas and problems of inheritance-related access control. Some of the points will not be too familiar to most readers; also, the distinction between private and public inheritance will be essential in the ensuing analysis of MI. Some language modification proposals will appear already here.

Section 4 treats the simpler form of MI, so-called independent multiple inheritance. I will claim that the current C++ rules have a severe defect that affects already this case. Section 5 discusses the more complicated case of MI, so-called fork-join inheritance. It is essentially more complex than would appear from the typical literature examples containing four or five classes. Not surprisingly, here I will present the largest number of defects, problems, and solutions.

The most important points will be emphasised in the form of six theses and one rule in boldface. In fact, a hasty reader may look up just these points and perhaps the conclusions. The rule, which I believe to be the most significant single contribution of this paper, will be found in

[1] Fifth European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15□19, 1991.

Subsection 5.4. It is accompanied a little later by two restrictions.

The paper tries to be as self-contained as possible in its reasoning. However, there are more numerous and more detailed references to existing literature than may be conventional in this journal. Those readers who are not interested to compare the various papers in detail can safely ignore the references.

2. The *if* and the *how* of multiple inheritance

2.1. *If:* the main points of Cargill and Waldo

Cargill [1991a §4] describes the complexity of current C++ inheritance:

> [There are] six variants of inheritance: a choice of three access levels for each inheritance relationship (public, protected or private), and another choice of whether or not each base class is virtual. The real expressive power of inheritance is delivered by just one of the six variants: public inheritance from a non-virtual base.

I disagree with the last statement, and will argue against it in a later section. I agree about the complexity (see §3.2 about *protected* base classes), and so did Waldo.

Cargill goes on to show how multiple inheritance further increases the complexity of the language. We have no argument here: Bjarne Stroustrup himself has been the first to admit and explain the basic complications. Note that already the choice between "virtual" and "non-virtual" base class mentioned above was introduced because of MI.

The other main argument of Cargill [1991a] is that no convincing examples of MI had been published. I had wondered a little about that myself, but reasoned that such examples would tend to be too long for typical journal and conference papers. However, I had heard people complain how impossible it was to utilise two independent, extensive class libraries such as OOPS (currently NIHCL) [Gorlen 1987] for "foundation classes" and InterViews [Linton & Calder 1987] for the user interface, within the same software system using C++ without MI.

Waldo [1991a] suggests that the MI examples cited by Cargill are unconvincing mainly because they are based on *implementation inheritance*. This is a very noteworthy point in my opinion. He presents an example with *interface inheritance* and "*data inheritance*", in outline. He convinces at least me that

(1) the example is not artificial, but programmers can often be confronted with similar situations, and
(2) the problem could be solved only in very contorted ways if MI were not available.

The true usefulness of multiple inheritance (in C++) has thus been demonstrated. However, to a sceptic this only means that MI cannot be dismissed off-hand; the tradeoff between its advantages and disadvantages still remains a matter of judgement.

Cargill's paper nevertheless makes a lot of sense, especially from its chosen viewpoint of practical programming. At several places it really does not criticise so much *multiple* inheritance, as the common tendency in OOP to apply inheritance even where simple aggregation would be more appropriate; I could not agree more.

2.2. *How:* the different cases

Waldo [1991a] seems to perceive Cargill as quite adamant against multiple inheritance; I have a more open-minded impression. Cargill [1991a §6] actually sketches a class structure from which MI could not be eliminated without serious distortion, and says:

> If MI were widely used in this manner in real programs, my thesis would collapse.

The main idea is that at least some virtual function of each base class is redefined in the derived class so that it calls a virtual function of *another* base class. Interestingly, Waldo's example is totally different from this structure.

We can distinguish between two main forms of MI. By *"independent multiple inheritance"* (IMI) we mean that parallel superclasses have no common ancestors, or that there is only one derivation path connecting a class to an non-immediate base class[2]. The opposite case we will call *"fork-join inheritance"* (FJI), as coined in Sakkinen [1989] in analogy with the forking and joining of parallel processes. I have not seen any other compact term for this phenomenon in the literature.

Cargill [1991a §8] briefly admits the need for IMI, motivated by multiple independent class libraries (cf. §2.1):

> We may discover that MI is indeed useful, but that virtual base classes are unnecessary.

[2] This term is obviously used in a slightly more general meaning in Stroustrup [1989b].

He has later [1991b] paraphrased this unambiguously:

FJI is not useful and therefore the virtual base question is moot.

Waldo does not take a stand on this issue in the article [1991a], but he has later [1991b] clarified his position to be rather opposite to Cargill:

[...] my point might show that virtual base classes are the only candidates for multiple inheritance.

In IMI there is no difference between virtual and non-virtual base classes. Therefore, Cargill's original statement could have been interpreted thus:

FJI is useful, but only with non-virtual base classes;

at least out of context. This was certainly not the intention. Cargill does not give any semantic reasons against virtual base classes; his goal is to reduce the complexity mentioned in the previous section. We have here a similar tradeoff situation as with MI in general. Neither Cargill's nor Waldo's example contains fork-join inheritance; thus the "no supporting evidence" argument remains valid. I will later try to suggest some reasons in favour of FJI, but they will not be very decisive.

Baclawski [1990] calls MI with non-virtual base classes 'multiple independent inheritance'. This includes both IMI as defined above and non-virtual FJI. Baclawski regards this as a peculiar variation of MI.

Another recent paper with some highly interesting points on MI — although I will not agree on all of them — is Snyder [1991]. It has been written with the purpose of describing the most essential properties of C++ in terms of a supposedly language-independent model. Snyder regards non-virtual FJI as an unusual corner case of the language; he did not bother to make his general model so complex that it could account for this situation. FJI with virtual base classes is not treated specifically in his article.

3. Inheritance and accessibility

3.1. Access levels of class members

There were originally only two alternative accessibility levels for both class members and base classes, `public` and `private` (and actually no explicit specifier yet for private accessibility). The intermediate level, `protected`, was added for members only in C++ Release 1.2 from AT&T.

C++ checks for name clashes between inherited and non-inherited members (and between members inherited from different base classes) *before* applying access controls. I think that this is the wrong choice, the rationale in Ellis & Stroustrup [1990 §11.3c] notwithstanding:

> Making a name public or private will not quietly change the meaning of a program from one legal interpretation to another.

The flaw in this reasoning is that changing the access level of a class member is a modification of the class's *interface*, which should always be expected to affect the clients of the class and should not be done too lightly.

On the opposite, adding or renaming a private member is only a matter of the class's *implementation* and therefore should not concern clients at all. As things are in C++, private members of a class cannot be of any benefit to derived classes (and outside clients) but can cause harm to them. The same holds for members of private base classes. The special case of `virtual private` functions will be discussed in §3.4.

When `protected` class members were first introduced, their definition was simple and sensible [AT&T 1986]. A protected member `m` defined in a class `A` was accessible both to class `A` itself and to any class `B` directly derived from `A`; unless the derivation was private, the accessibility of `m` in `B` was the same as if `m` had been a protected member of `B` itself.

With Release 2.0, the meaning was subtly changed. Even if a protected member `m` of class `A` is accessible to some descendant class `C`, member functions of `C` are now allowed to access the `m` part of an object only if that object is statically known to belong to class `C` or a descendant of `C`.

The rationale in Ellis & Stroustrup [1991 p. 254] says:

> [The original rule] would allow a class to access the base class part of an unrelated class (as if it were its own) without the use of an explicit cast. This would be the only place where the language allowed that.

'Unrelated' in the quote seems queer since the classes have a common ancestor and only the common part would be accessed. Otherwise this argument makes some sense — I assume that mentioning explicit casts is not meant to imply that they could be used to bypass the new restriction.

The example on p. 255 has a class `Account` with derived classes `checking_account` and `AutoLoan_account`, and it is argued:

> [`Account`] has information common to all kinds of accounts, including the account balance, so a friend of `Account` can walk the list of all `Accounts` and tell [the balances]. [...] The member functions of `checking_accounts`, however, should not be able to access the balance in an `AutoLoan_account`. The restriction prevents that.

This also makes some sense, but not completely. If a friend function is supposed to be able to treat the balance in all possible classes derived from `Account` correctly, why should a member function of a derived class be unable to do the same? I would much more want to prevent other people's account *instances* from accessing the balances of my accounts (of whatever kind), but that is not possible in C++.

In my opinion, this complication is a display of paternalism contrary to the general philosophy of C++[3] (cf. §3.7). It can prevent some obscure programming errors, but also makes a potentially much larger number of perfectly sound and useful pieces of code illegal.

3.2. Modes (access levels) of inheritance

There is an uncertainty on whether `protected` base classes should be possible in current C++. Ellis & Stroustrup [1990] is inconsistent on this point, with most clues leading to a negative answer. At least the Hewlett-Packard C++ translator/compiler does not accept `protected` base classes. On the contrary, Stroustrup [1991] makes it clear that protected base classes are indeed meant to be possible (in some future version?), and also describes their semantics.

Bjarne Stroustrup's books and articles try to make a clear conceptual distinction between public and private inheritance (derivation). For a long time, I was in doubt about how protected inheritance could logically fit into this picture. Now I think that one can regard protected inheritance as a restricted case of public inheritance, in that both these modes are *transitive*: any class in an inheritance hierarchy can access all its non-immediate base classes. For the purposes of this paper, it suffices to speak of public inheritance, and the results should be applicable to protected inheritance as well.

Private inheritance, in contrast, is *intransitive*: every class can access only its immediate base classes. This is the kind of inheritance that has been recommended in earlier work of Alan Snyder [1987]. It is akin to "incidental inheritance" in Sakkinen [1989], while the public inheritance of C++ corresponds to "essential inheritance".

Public inheritance is supposed to imply more or less an *is-a* relationship, which must be transitive. Already because of its intransitivity, private inheritance does not imply any *is-a* relationship. In private (single) inheritance, the role of the base class can be very similar to the role of a *representation* in CLU [Liskov et al. 1981].

[3] I would still prefer a more paternalistic general philosophy!

C++ differs from the majority of object-oriented languages in that the data members of an object are directly contained in the object, independently of their type. In most other languages, an object can contain only *pointers* to other objects; this is called "reference semantics". Because of this property, private inheritance is much less different from aggregation (i.e., a private base class from a data member) in C++ than in those other languages. One might suggest that private (and protected) inheritance be eliminated: that would simplify the language quite a bit.

I can agree with Baclawski [1990][4] and Cargill [1991a] that public inheritance is the more important case and should preferably have been the default. However, I cannot totally agree that private inheritance is *only* aggregation with some syntactic sugar, although I had suggested in Sakkinen [1989] that *incidental* inheritance could be replaced by aggregation and three simple constraints. Baclawski is wrong in claiming that private inheritance in C++ does not support late binding; I had not quite realised that earlier, either. In reality, a virtual function of a base class can be redefined even in a privately derived class — with less restrictions than I would like (§3.4) — and its invocations from other functions of the base class and the derived class will be late-bound.

Because `private` derivation does conserve this essential property of object-oriented inheritance, late-bound self-reference, I am at the end not willing to have it removed from C++, in spite of the complications. It is also such a traditional feature that its elimination would break too many pieces of existing software. `Protected` derivation could easily be omitted now as it is just being introduced. On the other hand, it adds only little complexity, and can probably be useful in some situations. It is also more orthogonal that the sets of possible access levels are the same for members and for base classes.

3.3. Some definitions

We will now define some terms that will be needed a little later. The more generally used term 'inheritance' has already been used as a synonym of 'derivation', which is more common in C++ literature. The words *'ancestor'* and *'superclass'* will be used to mean immediate or non-immediate base class, and likewise the words *'descendant'* and *'subclass'* to mean immediate or non-immediate derived class. (These meanings

[4] Baclawski [1991] says that this assessment was originally made by Stroustrup himself.

correspond to 'proper ancestor' and 'proper descendant' in Meyer [1988].)

The accessibility of ancestors to descendants is probably intuitively clear to those readers who know enough about C++ to have bothered to read this far. The following definitions will make our concepts precise enough. For the purposes of MI we need to think about the accessibility of *paths*, while the accessibility of (ancestor) classes would be sufficient for SI.

Let us regard the *inheritance graph* of a "closed"[5] collection of C++ classes as a *labelled* directed graph, where each edge is directed from derived class to base class and labelled with its access mode and sharability (virtual or non-virtual). An inheritance graph is always a directed acyclic graph (DAG), but *not* in general a lattice, contrarily to what is often incorrectly claimed in the object-oriented literature. We will call any path (sequence of nodes and adjoining edges) in this labelled DAG a *derivation path*.

A derivation path will be called *transitively accessible* if it contains no edge labelled private (which includes the special case of a zero-length path), *intransitively accessible* if only the first edge is labelled private, and *inaccessible* otherwise. A *class* `A` is called accessible to a class `B` if there is at least one accessible path from `B` to `A` (note the direction: `B` is either `A` itself or a descendant of `A`), and inaccessible otherwise. When there is a danger of ambiguity, we can use the longer word 'inheritance-accessible' about classes.

What does this mean in practice? To any class `B`, `B` itself and its public (and protected) ancestors are transitively accessible. The direct private base classes of `B` and their public (and protected) ancestors are intransitively accessible to `B` (unless also transitively accessible by another path). The private ancestors of any ancestor class are inaccessible to `B` (unless accessible by another path). — The distinction between accessible and inaccessible ancestor classes will be crucial in the sequel. The distinction between transitive and intransitive accessibility will be also be needed, but much less often.

The reader should convince him/herself that the accessibility gained by a descendant due to inheritance in C++ indeed corresponds to the above definition. We expressly exclude `friend` accessibility (which is intransitive like private inheritance) from these definitions, for instance because friend relationships do not affect late binding. The fact that a descendant may gain better access to an ancestor by being declared also a friend should just be remembered.

[5] For every class in the collection, all its ancestors must also be in the collection.

For every single class `C` we define the inheritance graph *of* `C` as consisting of all derivation paths whose first node is `C`. For every instance of `C` there is a *subobject graph* corresponding to the inheritance graph, where each subobject node is labelled with the name of its class. Each subobject contains the non-inherited non-static members of its class. All the graphs in Ellis & Stroustrup [1990 §10] are subobject graphs: the nodes are subobjects and not classes.

The correspondence between *paths* in the two graphs is one-to-one. However, a class in the inheritance graph may correspond to more than one node in the subobject graph, depending on the sharabilities. That is one of the main issues of §5. For that discussion we will need one more definition: the *complete subobject* corresponding to a node N in the subobject graph shall be the subgraph reachable from N, i.e. consisting of all paths whose first node is N.

3.4. Problems of private inheritance

There is a flaw in the access control principles that only affects virtual functions. Namely, a derived class can *redefine* any virtual function of any ancestor class, even those that it cannot *invoke*. We illustrate this with an example:

```
class Top {
public: /* or protected: */
     virtual void work();
     ...
};
class Middle: private Top {
     ...
};
class Bottom: /* any mode */ Middle {
     ...
     virtual void work();
     ...
};
```

Example 1.

All calls of `work` in member functions of `Top` without explicit class qualification will invoke `Bottom::f`, within a `Bottom` object. Private derivation thus does not isolate the classes `Top` and `Bottom` from each other.

I propose in a little oversimplified way:

Thesis 1: A descendant class must not be able to redefine a virtual function of an inaccessible ancestor class.

In the next subsection I will suggest the possibility of purely static overriding, however. It may seem that we have here only a question of taste instead of a true anomaly in the current C++ rules. Stroustrup [1991b] says that he did not want to be paternalistic. However, without this restriction it is not possible to avoid the "exponential yoyo problem" (§5.6).

There is an interesting consequence of Thesis 1. It obviously makes Example 1 illegal, or at least prevents late binding from `Top` to `Bottom::work`. That does not change if a redefinition of `work` is added to class `Middle`. Therefore, `Middle::work` cannot be virtually redefined further. according to the basic principle expressed in Ellis & Stroustrup [1990 p. 205]:

For virtual functions, [...] the same function is called independently of the static type of the pointer, reference, or name of the object for which it is called.

Corollary 1: A virtual function of class `C` whose original definition is inherited from an intransitively accessible ancestor class, must not be further redefined in descendants of `C`.

Thesis 1 was originally formulated thus: "A descendant class must not be able to redefine a virtual function that it cannot access." Under this rule it would not make sense to declare a member function `private virtual`. Bjarne Stroustrup convinced me that there can be a scenario in which the virtuality of a private function is useful, and that virtuality should be independent of the access level. He could *not* convince me that virtuality should be independent of derivation modes.

We thus split the accessibility of a virtual function into *invokability* and *redefinability*. The former is initially defined by the access level and the latter by the virtuality; for an inherited function, both are affected by the mode of derivation.

The usefulness of a private (instead of protected) virtual function is rather marginal. Let us examine a simple example, enhanced from Stroustrup's [1991b] by the addition of the intervening class `B`:

```
class A {
private:
     virtual void f();
public:
     virtual void g() { ...  f();  ... }
     ...
};
class B: public A {
     // no redefinition of f
     ...
};
class C: public B {
     ...
     virtual void f();
     ...
};
```

Example 2.

Obviously the member functions of class `B` cannot invoke `f`; this restriction can be desired in some situations. There is no declaration in current C++, however, that would prevent class `C` from invoking `f`. On the other hand, `C::f` cannot invoke `A::f`, nor can a redefinition of `g` in class `B` invoke `f`, both of which would typically be wanted. The code of `A::f` should then instead be duplicated in `C::f` or `B::g`, which is against the object-oriented reuse principle.

3.5. A proposed solution

We must search deeper in the foundations for a totally consistent solution. It appears that C++ has no mechanism nor even term for handling a "family" of virtual functions, i.e. the "most base" function together with its all redefinitions. This is one of the most important concepts in Snyder's [1991] object model: he uses the term 'operation' in this specific meaning. One of the most difficult questions in that paper indeed is: "What are the operations?" Pointers to member functions are a partial answer (see §4.1, §4.2).

In current C++, every member[6] of a virtual function family is

[6] Here 'member' is in its ordinary meaning, not in the somewhat confusing C++ meaning (component).

considered a completely independent function, except for late binding. Therefore also the invokability of a redefined function may be different from that of the original one. I suggest that a redefinition should *never* change the invokability of a virtual function. Of course, the mode of derivation may *lower* the invokability.

Thesis 2: The invokability of a redefinition of a virtual function in the redefining class should always be the same as that of the inherited function.

Note that according to Thesis 2, the function `f` in Example 2 would not be invokable even from class `C`, even though it is redefined there. This may sound surprising, but is in a way more consistent than the existing situation, where class `B` cannot invoke `f` although both its superclass and subclass can. Further, I really think that the cases in which a virtual function should be private are very rare.

There appears to be a nice way to satisfy the thesis while solving another, related problem. The meaning of virtual function declarations is currently quite context-dependent. An explicit `virtual` specifier can mean either that a new virtual function family is being defined or that an inherited virtual function is being redefined (this ambiguity was not allowed originally). The omission of `virtual` can mean either that an inherited virtual function is being redefined, an inherited non-virtual function is being statically overloaded, or a new non-virtual function is being defined. It would be much better to have a distinct keyword for the redefinition, as e.g. in Eiffel (cf. §4.2).

According to Thesis 2, the invokability of a virtual function redefinition would not be affected by the member access specifiers; this is similar to `friend` declarations. A syntactically distinguished virtual function redefinition would be elegant also from this viewpoint. In Example 1, the following could be added to the definition of class `Middle`:

```
redefine void work();
```

Allowing the static (non-virtual) overriding of an inaccessible virtual function could be possible as an extension to the current C++ rules; it only makes sense because of the other suggestions. The function `Bottom::work` in Example 1 would then be legal, but only a static overloading of `Top::work` and would not belong to the same family, although itself virtual. In contrast, it will be lightly suggested in §3.7 that the non-virtual overriding of *accessible* members should not be allowed — a restriction to the current rules.

It has been conceded in §3.4 that the scope of virtual redefinability of a member function can sometimes be larger than that of invokability. Probably more often one would like to restrict the scope of redefinability to be *smaller*, i.e. to prevent further redefinitions from some class downward but retain invokability. BETA [Madsen 1987] is one language that offers such a possibility. Let us modify Example 1 again a little, with some tentative syntax:

```
class Middle: public Top {
      ...
      freeze void work();
      ...
};
```

Note that the derivation from `Top` was changed to public. Class `Bottom` would then be able to invoke but not to redefine `work`.

Adding new keywords to C++ is always a bit dubious, or at least requires very good reasons. This is a general problem in languages in which keywords are in the same space as programmer-invented names. One could manage without new keywords by replacing `virtual`, `redefine`, and `freeze` in the suggestion by, say, `virtual >`, `virtual =`, and `virtual <`, respectively. Unfortunately, one reviewer was afraid that this alternative could cause trouble for parsing.

3.6. Virtual base classes

Although the problems of `virtual` base classes would logically belong to the section on fork-join multiple inheritance, they are so central in the discussion that we may note some points already here. To begin, I think it has been an unfortunate choice to overload the term 'virtual' with this meaning, remotely related to the original one; for instance 'shared' would have been a better word. Recall that the meaning is [Ellis & Stroustrup 1990 p. 200]:

> A single sub-object of the virtual base class is shared by every [derived][7] class that specified the base class to be virtual.

As explained by Stroustrup [1987, 1989] and Ellis & Stroustrup [1990 §10], virtual base classes are a little more difficult and more costly to implement than non-virtual ones. Even then they are subject to the additional restriction that a pointer to a virtual base class cannot be cast into a pointer to a derived class. Casts from base to derived class are, however, extremely risky in C++ even in the cases where they are allowed, and should be avoided. The reason is the lack of run-time type information in objects — in my opinion, insufficient object orientation.

[7] It actually reads 'base' in the book, but that must be a simple clerical mistake.

Stroustrup admits that writing virtual functions can be trickier in the presence of virtual base classes. The example in e.g. Ellis & Stroustrup [1990 p. 201–202] shows that one must often write another, protected and non-virtual function for descendant classes to call; otherwise the function in some base classes may get invoked more than once. Cargill [1991a] dislikes the complexity caused by possible "sideways" redefinitions of virtual functions with virtual base classes. We will treat this question in §5.5–5.6 and find that FJI with non-virtual base classes can actually lead to much more anomalous situations.

The virtuality of base classes must be considered in object construction, too. It is written in Ellis & Stroustrup [1990 §12.6.2]:

A *complete object* is an object that is not a sub-object representing a base class. Its class is said to be the *most derived* class for the object. All sub-objects for virtual base classes are initialized by the constructor of the most derived class.

This is obviously necessary in some situations, but in this formulation it is a too sweeping requirement, which may cause the effects of virtual derivation to propagate too far. It should be refined.

Consider this example:

```
class A {
public:
     A(int);
     ...
};
class B : public virtual A {
public:
     B(int i) : A(i) { ... }
     ...
};
```

Example 3.

Now all constructors of every class derived from `B` directly or indirectly, even by single inheritance, must explicitly invoke the constructor of `A`! The rule also requires an ugly exception to the accessibility rules of class members: the constructors of a virtual base class are accessible both to the virtually derived class and all its descendants, ignoring all access specifiers.

The effect of declaring a base class virtual is too global also in the sense that there is only one subobject within a complete object that corresponds to *all* occurrences of the same class as a virtual base. Although this seems simple and logical at first, it is less logical in some more complicated inheritance graphs. I will suggest an essential modification to this principle in §5.4.

Thesis 3: The effects of declaring a derivation `virtual` should not propagate too far in the subobject graph.

3.7. Non-virtual overriding and overloading

The philosophy of name scoping between superclasses and subclasses in C++ is the same as in Simula™: the scopes are regarded as if they were lexically nested. Variables and functions of the derived class can thus non-virtually override (hide) those of the base class.

This philosophy originated when there was yet no mechanism similar to the access modes of C++, and was then a sensible way to decrease unnecessary interference between super- and subclasses. However, now that the different access modes exits, it would be more natural to regard the accessible members of a base class to be in the *same* scope as the members of the subclass itself.

In consequence, I suggest that the overriding of accessible non-virtual base class members should not be allowed in a derived class; it is not very useful but can cause treacherous errors. Especially the difference between virtually and non-virtually overridden member functions is quite subtle. The overloading of non-virtual member functions would of course not be forbidden if the types of the explicit arguments are different.

The previous suggestion is more or less tentative; the following one is more serious. The hiding rule has been extended (I suppose in Release 2.0) so that defining or redefining a member *function* in a class also hides any inherited, overloaded member functions with the same name. This is another paternalistic rule (cf. §3.1) that should be retracted. It tries to prevent some errors but causes much more nuisance to perfectly good programming.

Ellis & Stroustrup [1990] give two examples as a rationale for the rule (§13.1, p. 310□312). The first example is based on the thinking that a client of a derived class should not need to be aware of the public functions of public base classes. In my opinion, this is contrary to the very idea of public inheritance. A work-around for accessing a hidden inherited function is presented:

```
class X1 {
public:
     void f(int);
};

// chain of derivations X2 .. X8

class X9 : public X8 {
public:
     void f(double);
     void f(int i) { X8::f(i); }
};
```

Example 4.

Having to write a lot of such auxiliary functions can be an unnecessary pain in the neck for programmers. If the hiding rule were kept as the default, at least one should have some more convenient means to escape it, such as:

```
     reveal void f (int);
     reveal f;
```

The latter declaration would reveal all overloaded, inherited variants of `f`. Even better, the hiding rule should not be the default, but applicable by explicit `hide` declarations analogous to the above `reveal` declarations.

The second example of Ellis & Stroustrup [1990 p. 312] is based on an assignment operator:

```
struct B {
     void operator= (int i)
     ...
};
```

If this operator were not automatically hidden in subclasses of `B` by the default assignment operator (if nothing else), it would probably cause an incomplete assignment when applied to a subclass object. — Here the hiding rule tries to protect against sloppy programming, but succeeds only half way: an invocation through a pointer of type `B*` will cause the "incomplete" operation to be performed anyway. The operator should absolutely be declared virtual in the first place and redefined in the appropriate derived classes.

The above kind of `reveal` declaration would also be a better way than the existing one [Ellis & Stroustrup 1990 §11.3] for restoring the original accessibility of selected members of private (or protected) base classes. A small but admitted defect of the current method is that overloaded member functions with the same name cannot be treated separately. To take the book's example:

```
class X {
private:
     f(int);
public:
     f();
};
class Y : private X {
public:
     X::f;          // error
};
```

Example 5.

There is no way to restore the accessibility of `X::f()` to `public` in `Y` because `X::f(int)` is `private`.

4. Independent multiple inheritance

4.1. Problems in current C++

Simplifying things a little, we can say that the prime conceptual problem of multiple inheritance are *horizontal* name clashes, i.e. those between parallel superclasses (base classes). Independent MI is a simple and unproblematic case *in principle*: any such name clashes can be regarded as purely accidental and resolved by explicit class qualification.

Name clashes between *data members* indeed cause no big trouble in C++: one must just explicitly qualify the member name with the appropriate class name when referring to it in a derived class. Unfortunately, name clashes between *function members* cannot always be handled adequately. Let us consider an example from Stroustrup [1991 §13.8]: the two totally unrelated classes `Window` and `Cowboy` both happen to have a function called `draw`, but their meanings are obviously quite different.

```
class Window {
     // ...
     virtual void draw();
};
class Cowboy {
     // ...
     virtual void draw();
};
class CowboyWindow : public Window, public Cowboy {
     // ...
};
```

Example 6.

There is a similar example in Snyder [1991 Fig. 8b]. Snyder sees here an anomaly in C++: that there is no way to denote either `Window`'s or `Cowboy`'s `draw` in the lexical context of `CowboyWindow`, in a way that would not suppress virtuality and would be resilient to a later evolution of the class hierarchy. Indeed, a simple class qualification (`Cowboy::draw` or `Window::draw`) may need to be changed if `draw` is later redefined in `CowboyWindow`, or if some new class redefining `draw` is interposed in the inheritance graph between `Cowboy` and `CowboyWindow` (or `Window` and `CowboyWindow`).

Actually there *is* a solution to this problem, by using "pointers to members"[8] — a difficult new feature which happens to be discussed in Snyder [1991] as well. Specifically, if `cw` is an instance of `CowboyWindow`, one could use

```
(cw.*(&Cowboy::draw)) ()
```

to invoke the first function, and

```
(cw.*(&Window::draw)) ()
```

to invoke the second function. As Ellis & Stroustrup [1990, p. 157] confess, "the syntax isn't the most readable one can imagine", but it should work exactly as Snyder wanted. — Even this solution does not work for *non-virtual* functions. However, if the suggestion of §3.7 to forbid non-virtual overriding were accepted, ordinary class qualification would suffice for them.

This whole example would have been illegal C++ and therefore moot according to the rules of Stroustrup [1989 p. 379]: a horizontal name conflict between *virtual* functions was required to be resolved by a redefinition in the derived class. This rule has obviously been lifted, sensibly

[8] Again a term that I do not really like, because those are offsets rather than pointers.

enough.

In my opinion, the real conceptual fault appears first when one would like to redefine one of the inherited `draw` functions. Namely, it is impossible to redefine `Cowboy::draw` and `Window::draw` separately in `CowboyWindow`: a redefinition will *unify* the functions. The situation is the same whether the functions are virtual or not. This is absolutely wrong if we assume that name clashes between independent superclasses really are accidental. In Example 6 it is impossible to imagine a function that could be a sensible common specialisation of the two `draw` functions. An analogous situation in Snyder [1991 Fig. 8a] is only called "challenging" there, in the sense of its modelling being non-obvious.

Thesis 4: The language must not force functions inherited from mutually unrelated ancestors to be unified in a common descendant class.

It is interesting to note that Ellis & Stroustrup [1990 §10.11c] recognise this problem, but do not seem to consider it important. They actually write:

> The semantics of this concept are simple, and the implementation is trivial; the problem seems to be to find a suitable syntax.

Syntax concerns seem a poor excuse for leaving the problem unsolved, especially as the syntax of C++ is not so wonderful anyway.

The anomaly of the above becomes still more obvious if we think of cases in which the two `draw` functions are *not* of exactly the same type. If they differ only in their return type, then it is not possible to redefine either of them in `CowboyWindow`, as far as I can infer from Ellis & Stroustrup [1990]! If they differ in the types of arguments, then they remain separate even in `C`, but if only one is redefined the other becomes hidden (§3.7).

4.2. Different solutions

Stroustrup [1991 §13.8] presents a work-around for redefining the `draw` functions in Example 6. `CowboyWindow` cannot be derived directly from `Cowboy` and `Window`, but auxiliary intermediate classes and functions are needed. This is Stroustrup's solution, with trivial type errors corrected:

```
// ...
class WWindow : public Window {
     virtual void win_draw() = 0;
     void draw() { win_draw(); }
};
class CCowboy : public Cowboy {
     virtual void cow_draw() = 0;
     void draw() { cow_draw(); }
};
class CowboyWindow : public Window, public Cowboy {
     // ...
     void win_draw();
     void cow_draw();
};
```

Example 7.

This method would work just as well even if the two `draw` functions had different result types. However, it looks a little awkward, adding complexity to the class structure. The awkwardness becomes worse if we suppose that the classes `WWindow` and `CCowboy` should be reusable, and that there may be more than one pair of colliding functions that could be redefined. In order not to require every derived class to redefine every function, `win_draw` and `cow_draw` should actually not be defined as *pure virtual* (`= 0`), but rather like this:

```
     virtual void win_draw() { Window::draw(); }
```

More importantly, programming tools such as class browsers probably cannot tell the programmer that in order to get a redefinition of `Window::draw` in classes derived from `CowboyWindow`, it is the function `win_draw` that must be redefined.

One way to avoid the problem of mixing up unrelated functions would be the mechanism of *"titles"* [Sakkinen 1990]. I had originally thought it out earlier, and noted when the "pointer to function member" concept was added to C++ that there was a strong similarity. A bit paradoxically, although families of virtual functions are not well defined in C++ (§3.5), pointers to them now exist. As Snyder [1991 p. 13] puts it:

> Pointers to class function members correspond exactly to operations in our model of C++: such pointers *cannot* distinguish between individual methods for the same operation [...]

Simplifying to the most essential for this case, a title would have a meaning lying between a class qualification and a pointer to function member. To try some concrete syntax, `A..f` would mean "the most specific overriding of function `A::f`". The most important difference to `*(&A::f)` would be that this construct could be used also in redefinitions. In the above case, one would use just `A..f` if redefining the function lexically within the definition of a derived class `C`, and `C::A..f` outside class definitions.

Using the title could automatically take into account even non-virtual overridings (previous subsection). If all the suggestions in §3.5 were realised, titles could be needed for the opposite purpose: to disambiguate *vertical* name clashes between virtual functions already in single inheritance. Member function pointers are also adequate for that task, though, because the need would appear only in invocations, not definitions. — Neither of these two uses would be relevant if non-virtual overriding of accessible members were forbidden (§3.7).

Finally, in spite of what was said above, in some cases one might *want* to unify two functions inherited from different superclasses. This could be done by equating their titles (possible syntax is left to the reader's imagination), even if the original names were different. Of course, the argument and result types of both functions should be identical.

Note that in Eiffel™ [Meyer 1988 §11.2], inherited *features* (Eiffel terminology, equivalent to 'members' in C++ terms) with identical names can be redefined separately:

```
class Cowboy feature draw ... end
class Window feature draw ... end
class CowboyWindow inherit
      Cowboy rename draw as cow_draw redefine cow_draw;
      Window rename draw as win_draw redefine win_draw;
   ... end
```

Example 8.

The suggestion for C++ mentioned in Ellis & Stroustrup [1990 §10.11c] is totally analogous to this. The advantage of my "title" solution is that it would not be necessary to invent new names for the overriding functions; although even in Examples 7 and 8, *both* were renamed only for the sake of symmetry.

4.3. Private multiple inheritance

The following section will present quite a lot of complications that are caused by the presence of private inheritance in C++, in contrast to most other object-oriented languages. Before that, let us have one more argument in favour of private inheritance.

As explained in Stroustrup [1991 §12.2.5], a typical simple use of IMI is to have one public base class and one private base class which serves as the implementation. The very first example of multiple inheritance in Meyer [1988 §10.4.1], already entitled "The marriage of convenience", is like this. If we disregard genericity, the example defines the class FIXED_STACK by inheriting both the *deferred* (Eiffel term for 'abstract') class STACK, which defines the interface, and the ordinary class ARRAY, which is used for the implementation. All features of STACK are *exported* by FIXED_STACK, corresponding to public derivation in C++, while no features of ARRAY are exported, corresponding to private derivation.

This example has often been frowned upon, but actually there is only one defect in it: in Eiffel nothing prevents an object of type FIXED_STACK from being assigned to a variable of type ARRAY, after which all ARRAY routines can be directly invoked. Here the private derivation of C++ has an advantage over Eiffel: in C++ it would not be possible for clients to implicitly convert a pointer to FIXED_STACK into a pointer to ARRAY. Unfortunately, an explicit cast is always possible; but explicit casts can cause even catastrophic effects in C++, so a wise programmer *writes* them only when necessary and *reads* them in existing code as warning signs.

5. Fork-join inheritance

5.1. The positive side

I tend to believe that many situations really demand FJI (with virtual base classes). On a conceptual level, such situations arise whenever we have several mutually independent classifications of the same domain. For instance: A vehicle is either a land, water, air, or amphibious vehicle; in another classification it is either private or public (not in the C++ sense!); in a third one it is powered by wind, man, animal, or engine. Any instance of a vehicle is nevertheless *one* vehicle, thus `vehicle` as a non-virtual base class would make no sense. — This example is perhaps not absolutely convincing, as it might be modelled adequately without using inheritance at all.

An interesting programming style has been suggested by Paul Johnson [1990] as *fine grain inheritance*, although I have not seen examples clearly showing its advantages. Very briefly, it means that every class should define a minimal coherent set of features, and a typical, conventionally designed class should be broken into a number of smaller classes related by (multiple) inheritance. Fine grain inheritance obviously requires a high degree of FJI with virtual public base classes.

One very simple example of public fork-join inheritance is presented by Stroustrup [1991 §6.5.1]:

```
class link { ... };
class task: public link { ... };
class displayed: public link { ... };
class satellite: public task, public displayed { ... };
```

Example 9.

With this definition there will be two separate `link` subobjects in each `satellite` object. If `link` were declared a *virtual* base class of `task` and `displayed`, there would be only one `link` subobject.

In Sakkinen [1989], I criticised Eiffel because, contrarily to C++, its rules would not guarantee the *integrity* of `link` subobjects in the above case. Depending on how the class `satellite` were defined, part of the components of `link` might well be shared and the rest duplicated. This happens on purpose in the "intercontinental drivers" example of Meyer [1988 §11.6.2]. The same danger seems to exist in several other languages that support MI.

Bertrand Meyer [1990] in turn has criticised the C++ principles on an issue slightly different from subobject integrity. According to his reasoning, there is no sense in having `task` and `displayed` decide about the sharing or duplication of `link`, since it does not affect them but only `satellite`. I continue to disagree with Meyer even on this point: it can be very important for the class `task` to know whether it has a `link` part to itself or is prepared to share it with any other, unknown class derived from `link`.

The question that I had forgotten to pose in admiring the MI principles of C++ [Sakkinen 1989] was: While C++ is in this respect more disciplined than e.g. Eiffel, does even the choice between virtual and non-virtual derivation independently of access mode, as allowed by C++, make sense conceptually and semantically? We will investigate this question, based in part on the analysis of Baclawski [1990].

In the following subsections, we suppose that a most derived class `C` is derived from several base classes by FJI. We examine how the subobject graph of `C` (§3.3) is related to its inheritance graph in various situations, and what restrictions are needed to guarantee the consistency and semantic feasibility of the inheritance structure.

5.2. Accessible base classes

Let us examine Example 9 further. The base class `link` is supposed to implement lists of objects: here a scheduler list of tasks and a display list. The derivations from `link` are therefore non-virtual, resulting in two distinct link subobjects in a satellite.

The inheritance in the example is public, which should imply transitive *is-a* relationships (§3.2). Since a satellite is a task and a task is a link, a satellite should also be a link. However, there are *two* distinct links in a satellite, thus one cannot say that a satellite *is-a* link. Indeed the rules of C++ will not allow us to convert a pointer of type `satellite*` directly to type `link*`, although such conversions are always possible in public *single* inheritance.

There will hopefully be no argument that one of the ubiquitous FJI examples in the literature of semantic databases is correctly modelled in C++ as follows.

```
class Person { ... };
class Student: public virtual Person { ... };
class Employee: public virtual Person { ... };
class StudentEmployee: public virtual Student,
        public virtual Employee { ... };
```

Example 10.

Of course the virtuality or non-virtuality of the immediate bases of `StudentEmployee` does not matter unless further classes are derived from it. The important thing to note is that the subobject structure from the viewpoint of `StudentEmployee` would not become different if its direct bases were made `private`. The ancestors would only become hidden from clients.

On these grounds I postulate a little imprecisely the following

Thesis 5: Accessible fork-join inheritance should always be virtual.

Note the more general qualification 'accessible' instead of 'public or protected'. In the pure case (i.e., no private derivations in the inheritance hierarchy) the thesis is sufficient and means that the subobject graph shall be isomorphic to the inheritance graph. In particular, an object shall contain exactly one subobject corresponding to each ancestor class. A precise formulation that is valid also for mixed derivations can be based on the definitions of §3.3: If class `A` is accessible to class `B` over more than one derivation path, an instance of `B` shall contain one complete subobject of class `A` that is common to all those derivation paths. (An instance can be either a complete object or a complete subobject.)

Now we have gotten into a conflict with Stroustrup's example, which looked perfectly natural at first sight. At least `link` simply cannot be a virtual base class there, according to the intended semantics. The only way to both preserve the inheritance graph and conform to the rule is to make `link` a private base class (of at least one of its immediate descendants). In fact, that would be reasonable also because otherwise it would be much too easy e.g. to put tasks on the display list.

Pondering Example 9 a little more, we note that it is questionable to make `link` a base class at all, instead of a data member. If a task, for example, should happen to need more than one link, it would not be possible to use inheritance. Declaring data members of type `link` in `task` and `displayed` would cause different problems: there would be no way for the `link` objects to refer to the objects in which they are contained. A suggested new feature of C++ that was obviously designed especially for cases like this, is *template class* [Ellis & Stroustrup 1990 §14; Stroustrup 1991 §8]. The class `link` could be declared as a template class taking a type name (here `task`, `displayed`) as its template argument.

One unconventional property of Eiffel is that *direct repeated inheritance* [Meyer 1988 §11.6.1] is allowed: one class may appear more than once in the immediate ancestor list of another class. This property is often regarded as suspicious, but the possibility of public FJI with non-virtual bases conceptually contains it as a special case, except for the subobject integrity problem in Eiffel (§5.1). To see that, simply imagine that the classes `task` and `displayed` in Example 9 declare no new members, but act simply as naming aids for the two `satellite` subobjects. Because of the renaming facility, such auxiliary classes are not necessary in Eiffel.

5.3. Inaccessible base classes

The special case of FJI where all derivations are either public or protected proved to be relatively simple in the previous subsection. The opposite pure case where all derivations are private is also simple. Private inheritance is intransitive, i.e. derived classes can access only their immediate bases. I postulate the counterpart to the thesis of the previous subsection, again as an imprecise slogan:

Thesis 6: Inaccessible fork-join inheritance should never be virtual.

In the pure case, this means that if `A` is an ancestor of `B`, an instance of class `B` shall contain a separate subobject of class `A` for each derivation path from `B` to `A`. In other words, the subobject graph shall be a tree. Obviously, totally private non-virtual FJI is no more complicated to understand and implement than private IMI. It would therefore need no very ambitious example of its utility in order to escape "Cargill's razor".

In the mixed case, Thesis 6 needs more adjustment than Thesis 5, because even a private path is accessible if its length is one. The complete formulation will be deferred to the following subsection but we refine the statement a bit here. As mentioned in §3.6, the virtuality of base classes is too strong or too global in current C++. What I want to achieve is that two separate subobjects cannot have a "hidden" common sub-subobject. Thus: If there is an inaccessible derivation path from class `B` to class `A`, the `A` subobject of an instance of `B` corresponding to that path shall be totally disjoint from all other `A` subobjects of that instance.

Combining the theses 5 and 6 we conclude that an explicit `virtual` declaration of a base class becomes superfluous. We could thus simplify C++ by omitting the `virtual` specifier for base classes. However, this holds only in principle; pragmatic needs will be suggested in §5.7.

5.4. Mixed cases

How do we get from the inheritance graph of a class `C` to the corresponding subobject graph in the general case, in which virtual (public or protected) and non-virtual (private) derivations can be arbitrarily combined? The non-transitivity of private inheritance makes the problem difficult.

I present a general rule, however, which I believe to result in consistent and understandable object structures even in very large and complicated class hierarchies.

Rule: Let *P* and *Q* be two derivation paths from class `B` to class `A`, having no common nodes except the end points. The paths *P* and *Q* will correspond to the same complete `A` subobject in an instance of`B` if and only if both are accessible. Otherwise the paths will give rise to two disjoint complete subobjects.

One consequence of the rule is that no class can access more than one subobject corresponding to each ancestor class. Multiple class qualifications (like `B::A::x`) will therefore never be necessary in order to uniquely denote inherited members. Stroustrup [1989] noted that such multiple qualifications would be useful under the current C++ rules, but they have not been introduced into the language in this purpose. Instead, when the nesting of class definitions was made meaningful (causing nested scopes) in Release 2.1 [Ellis & Stroustrup §9.7], this syntax was employed to refer to such nested definitions from outside.

In current C++ there are no restrictions on the legal inheritance graph of a class, except acyclicity. There can be at most one edge directly connecting any two nodes (classes), but that is a normal requirement for proper graphs. It appears that with the new rules as presented so far we can manage without additional restrictions on the basic graph structure. However, we will need further restrictions on the labelling (access modes) to assure consistency. The previous subsections already effectively removed the virtuality labels.

A problem point in the rule is revealed by the following anomalous example.

```
class A {};
class B : public virtual A {};
class C : private B, public virtual A {};
class D : public virtual C {};
```

Example 11.

The classes `A` and `C` are accessible to class `D`; even according to Thesis 1, virtual functions of `A` can be redefined in `D`, and such redefinitions are effective also for the `C` subobject of an instance of `D`. Because `B` is accessible to `C`, the redefinitions propagate also to the `B` subobject. On the other hand, class `B` is inaccessible to `D`; by Thesis 1 `D` should not be able to affect the virtual functions effective for its `B` subobject.

The contradiction is solved by the following

Restriction 1: If a class `C` has both a transitively accessible and an intransitively accessible derivation path to the same ancestor class, no further classes must be derived from `C`.

5.5. Virtual functions with virtual base classes

The late binding of virtual functions, often called “method lookup” in object-oriented literature, becomes complicated in the fork-join case. Let us first consider the case with virtual base classes, since according to my suggestions late binding would be relevant only there.

The little difficulty that was mentioned in §3.6 can be illustrated by augmenting Example 10 a little. This is equivalent to the example in Ellis & Stroustrup [1990 p. 201▫202]:

```
class Person {
public:
     virtual void earn();
     ...
};
class Student: public virtual Person {
public:
     virtual void earn();
     ...
};
class Employee: public virtual Person {
public:
     virtual void earn();
     ...
};
class StudentEmployee: public virtual Student,
          public virtual Employee {
public:
     virtual void earn();
     ...
};
```

Example 12.

Suppose that each version of `earn` has to call the inherited version(s) in addition to doing its "own job", as is very common. At least the classes `Student` and `Employee` must then define a separate, non-virtual protected function, say `own_earn`, that does the "own job", and every virtual function must call all relevant non-virtual functions:

```
void Student::earn() {Person::earn(); own_earn();}
void Employee::earn() {Person::earn(); own_earn();}
void StudentEmployee::earn() {
     Person::earn(); Student::own_earn();
     Employee::own_earn(); own_earn();
     }
```

For functions with a non-void result type (`virtual int earn()`), there is the additional problem of how to combine the results of all the different functions.

There is a slightly erroneous rule in Ellis & Stroustrup [1990 p. 235]:

> To avoid ambiguous function definitions, all redefinitions of a virtual function from a virtual base class must occur on a single path through the inheritance structure.

The rule could actually make sense, but it does not seem to describe current C++ correctly. Taken literally, it would make even Example 12 illegal. The intent evidently was:

> [...], if a virtual function from a virtual base class is redefined on more than one path through the inheritance structure, there must be one redefinition that dominates all others.

An *advantage* of the unlimited scope of redefinability of virtual functions in current C++ is that there is always a class where such a dominating redefinition can be done if an ambiguity must be resolved: at least the most derived class. Obeying Thesis 1, there could well be cases in which there is no single most derived class in which the virtual functions of a given ancestor class can be redefined. That would happen in Example 12 if the base classes of `StudentEmployee` were `private`. To avoid that, we must make an explicit

Restriction 2: In the inheritance graph of a class `C`, for any ancestor class `A` that is accessible to several descendants, there must be one among them to which all others are accessible.

The principle of dominance leads to the "sideways" inheritance mentioned in §3.6. Suppose that the redefinitions of `earn` are removed from `Student` and `StudentEmployee` in Example 12. Calls of `earn` in an instance of `StudentEmployee` will then always be resolved to `Employee::earn`, even when they are issued from functions of `Student` or by clients using a pointer of type `Student*`. Ellis & Stroustrup [1990 §10.10c] says:

> A call to a virtual function through one path in an inheritance structure may result in the invocation of a function redefined on another path. This is an elegant way for a base class to act as a means of communication between sibling classes [...]

Cargill [1991a §4] sees the disadvantages as more important (referring to a similar example):

> To understand the behavior of [`Student::earn()`] we must examine the entire DAG reachable by traversing from any class derived from [`Student`] to any virtual base class of [`Student`].

I must admit that both the advantages and the disadvantages of this method lookup scheme look important. However, since a *complete object* (§3.6) is primarily regarded as one single object, it seems logical that method lookup always begins this way, from the most derived class (i.e., the root of the complete object).

5.6. Virtual functions with non-virtual base classes

It has appeared to me that the method lookup in current C++ is in some ways more problematic in FJI with *non-virtual* base classes. In fact, I have not succeeded to find this case explicitly described in Ellis & Stroustrup [1990], nor in other books and papers! It is in most respects like independent multiple inheritance. Note that this situation can occur only in existing C++, not with my new rules.

Consider Example 12, modified so that all derivations are non-virtual. Let us pretend that this could be sensible, ignoring the too obvious real-world situation that a student employee is only one person. If `earn` were not redefined in `Student` and `StudentEmployee`, no direct "sideways inheritance" as with virtual derivation could occur. On the other hand, any invocation of `StudentEmployee::earn` would be a compile-time error; either `Student::earn` or `Employee::earn` would have to be selected statically.

Suppose again from now on that the redefinitions of `earn` are there. With non-virtual derivation we do not need the non-virtual auxiliary functions in every class, and `StudentEmployee::earn` need not worry about `Person::earn`. The situation thus looks much simpler:

```
void Student::earn()
     {Person::earn(); /* then the own stuff */ }
void Employee::earn()
     {Person::earn(); /* then the own stuff */ }
void StudentEmployee::earn()
     {Student::earn(); Employee::earn();
     /* then the own stuff */ }
```

Of course, if the result type of `earn` were non-void, there would still be the problem of result combination, as in the virtual case.

However, think about the case that `earn` is invoked by a member function of `Person`. This call then comes from the `Person` sub-subobject of either the `Student` or the `Employee` subobject, but it will cause `Person::earn` to be invoked on *both* `Person` parts! This is much more insidious sideways inheritance than in virtual derivation.

The effects can get even more interesting. Suppose that `Person::earn` calls another virtual function `work` of `Person`, and `work` gets redefined similarly to `earn`. It is easy to see that one call of `earn` from `Person` will cause `work` to be invoked twice on both `Person` subobjects! We might call this anomaly *"the exponential yoyo problem"* — the explanation follows.

The suggestive name *'yoyo problem'* was coined by Taenzer et al. [1989] to describe the following situation, translated from the terminology of Objective-C® into that of C++: Whenever virtual functions invoke other virtual functions of the same object (`*this`), the method lookup starts from the most specific class of the actual instance and proceeds upward in the inheritance hierarchy until a definition is found. The flow of control can therefore oscillate arbitrarily up and down and be difficult to follow if the hierarchy is deep.

The yoyo problem was identified in an environment with single inheritance. It obviously becomes more complex with multiple inheritance, as explained in the quote from Cargill in §5.5. Our new problem is clearly similar to the yoyo problem; it is exponential in the sense that the number of invocations gets multiplied by the number of inheritance branches on each down-and-up trip. However, the new case is clearly erroneous, whereas the original yoyo problem means only difficulties in understanding and debugging software.

The exponential yoyo problem does not appear naturally with virtual base classes. To see that, let us return to Example 12, including all function redefinitions. Let there be another virtual function `work` in class `StudentEmployee`, which is again redefined in all the other classes just like `earn`. If `Person::earn` invokes `work`, the invocation will be late-bound to `StudentEmployee::work`. That in turn will cause `Person::work` and each `own_work` to be called exactly once; no surprises. In order to cause multiple invocations, at least one of the `own_earn` functions must be expressly written to call `work`.

5.7. Further considerations

It was noted already in §3.6 that the liability of invoking the constructor for a virtual base class should not extend farther down in the inheritance graph than is logically necessary. Within the rules proposed so far in this paper, the following would be adequate: For every class `B` from which there are at least two *disjoint* accessible paths to an ancestor `A`, the accessible `A` subobject must be initialised directly by the constructor(s) of `B`; any initialisation of `A` specified by classes between `A` and `B` on the derivation paths will be ignored.

Alternatively, we could do this even simpler and specify that the initialisation caused by that immediate base class shall prevail which is mentioned first in the base list of `B`. This rule could be accompanied by a general ability of descendants to redefine the initialisation of any accessible non-immediate ancestor.

There are important exceptions to Thesis 5. First, there can sometimes be semantic reasons for public or protected inheritance with guaranteedly unshared subobjects. Second, the pragmatic viewpoint should not be ignored either. Presumably even in large inheritance graphs designed by programmers who know how to exploit the advantages of MI, there will be relatively few accessibly derived fork-join structures. Therefore one would not like to have the overhead of virtuality in all derivations.

Taking into account these factors and the tradition of the language, it is probably wisest to keep the `virtual` keyword but enhance its implications so that the rules prescribed in this paper will not be violated. Primarily, if class `A` is an immediate non-virtual base of `B`, it shall be illegal for any descendant class of `A` (including `B` itself!) to have an accessible path to `A` both over `B` and over some other immediate descendant. Even a `private virtual` declaration can then make sense, to assure that `B` remains a descendant of `A` independently of changes in its other inheritance relationships.

As a somewhat different situation, one might like to assure that a certain set of classes derived from a common abstract base class form a *taxonomy*, i.e. that they are both exhaustive and mutually exclusive on every derivation level. Baclawski [1990] says about the mutual exclusion of subclasses:

> Such a constraint cannot normally be enforced by a programming language [...] It is curious that taxonomies are often cited as a motivation for inheritance, yet few systems offer the means of constraining a set of types to be a taxonomy.

In the class dictionaries of the Demeter™ framework [Lieberherr et al. 1991], inheritance hierarchies are *forced* to be taxonomies.

A taxonomy does not completely prevent later fork-join inheritance, if multiple independent taxonomies of the same base class are allowed, as in the very rudimentary vehicle example of §5.1. The C++ Demeter system, contrarily to the Flavors Demeter system, at the time of writing does not allow such multiple taxonomies, nor FJI in general [Lieberherr 1991].

Frameworks such as Demeter that require the whole class structure to be defined before compilation may evidently cause additional overhead for the incremental addition of new derived classes. On the other hand, they can allow a much higher degree of *customisation* [Lea 90] and consequent run-time efficiency than more conventional ways of object-oriented software development. In particular, in such a framework it would not be necessary for programmers to specify non-virtual derivation for merely pragmatic reasons. Another advantage of Demeter is the automatic generation and propagation of member functions for classes. This could probably be extended to generate such typical virtual function patterns as presented in §5.5 for FJI.

6. Summary and conclusions

The discussion that this article is continuing began from the question *whether* C++ should support multiple inheritance or not. I now think that there is sufficient evidence to answer "Yes" to this question, at least for *independent* multiple inheritance. Therefore it appeared more important for me to study *how* MI should work.

During this work it became apparent to me that *private* inheritance, rather a speciality of C++, is a major cause of complexity. It also became apparent that private inheritance is semantically such an important and powerful tool that the complexity should be tolerated.

I found that there are some subtle inconsistencies in the current inheritance principles of C++ that affect already single inheritance. The most important flaws could be summarised by saying that private inheritance is not private enough. Independent multiple inheritance exposes at least one further defect: the unpreventable unification of member functions because of accidental name equality. Fortunately, all these flaws can be corrected by a small modification to the language.

Fork-join multiple inheritance was known to be the really complex case: it had caused the distinction between virtual and non-virtual base classes that was one of the main targets of Cargill's critique. I believe having made plausible enough that, on conceptual grounds, public (more precisely: accessible) inheritance should implicitly be "virtual" and private (more precisely: inaccessible) inheritance "non-virtual", and the language could therefore be simplified. Implementation reasons may make this principle too expensive for public inheritance in the cases when it is not actually needed. However, I suggested some possibilities for optimisation.

The "exponential yoyo problem" is presented in §5.6 as a *reductio ad absurdum* to show that at least the combination of non-virtual (duplicating) FJI with unrestricted redefinability of virtual functions, allowed by the current C++ principles, is unsound.

Although the suggestions of this paper would make FJI simpler and more logical, it still remains a complicated thing for both implementors and users of the language. The main suggestions should be applicable to several other object-oriented languages as well. We still need good examples of FJI to convince Tom Cargill and many others that it is worth the trouble. A major reason for the lack of good published examples of MI is that such examples tend to be large and complicated.

Very similarly, most of the problems and defects in the inheritance principles of C++ only become apparent when one considers inheritance graphs essentially more complex than those in the textbooks and reference manuals. Such complex situations have obviously not been sufficiently considered when the language has been designed. I also suspect that the design of multiple inheritance as well as several other feature has been too much driven by implementation considerations. In my opinion C++ is not sufficiently object-oriented; I hope to expound that in Sakkinen [1991].

I am now more optimistic about multiple inheritance than ever: it is a good thing in principle, *and* it can be done right; it just is not so simple as many of us have often thought. The current rules of C++ must urgently be revised, although a programmer who is convinced about the theses and other suggestions of this article can realise part of them by programming discipline. This holds for Thesis 1, Thesis 2 in most cases and Thesis 5 in simple cases; thesis 4 can be achieved by Stroustrup's work-around. However, C++ programmers should for the time being probably avoid MI as far as possible in order to avoid later trouble. Complex combinations of public and private, virtual and non-virtual fork-join inheritance are especially dangerous.

Acknowledgements

This research has been funded by the Academy of Finland, project 1061120 (Object-oriented languages and methods).

Several persons have helped me in discussions and correspondence, and by sending literature that I would not have had available otherwise. These include at least Bjarne Stroustrup (in spite of everything!), Kenneth Baclawski, Douglas Lea, Peter Grogono, Juha Vihavainen, Timothy Budd, Ian Holland, Antero Taivalsaari, Karl Lieberherr, Daniel Edelson, Alan Snyder, Tom Cargill, Peter Salus, Jim Waldo, and Jorma Kyppö. The comments of the anonymous reviewers were helpful, and have been taken into account.

The discussions in some Usenet newsgroups have been very stimulating and useful. The most important groups have been, of course, 'comp.lang.c++' and 'comp.std.c++' (although I have not had the time to follow these regularly any more), as well as 'comp.object'. The occasions to give a guest lecture largely about the theme of this paper both at the University of Tartu (Estonia) and at the University of Kuopio further helped me to clarify my thinking and expression.

References

AT&T C++ Translator Release 1.2 Addendum to the Release Notes, Murray Hill, NJ: AT&T Bell Laboratories, 1986.

K. Baclawski, The Structural Semantics of Inheritance, manuscript (submitted for publication), Boston, MA: Northeastern University, 1990.

K. Baclawski, private communication, 1991.

T. Cargill, Controversy: The Case Against Multiple Inheritance in C++, *Computing Systems*, 4(1): 69–82, Winter 1991a.

T. Cargill, private communication, 1991b.

M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Reading, MA: Addison-Wesley, 1990.

K. E. Gorlen, An Object-Oriented Class Library for C++ Programs, *Software — Practice and Experience*, 17(8): 503–512, August 1987.

P. Johnson, Fine Grain Inheritance and the Sibling-Supertype Rule, manuscript, Great Baddow, England: GEC-Marconi, 1990.

D. Lea, Customization in C++, *Proceedings of the 1990 USENIX C++ Conference*, pages 301–314, 1990.

K. J. Lieberherr, private communication, 1991.

K. J. Lieberherr, P. Bergstein, and I. Silva-Lepe, From objects to classes: algorithms for optimal object-oriented design, *Software Engineering Journal*, 6(4): 205–228, July 1991.

M. A. Linton and P. Calder, The Design and Implementation of InterViews, *USENIX C++ Workshop Proceedings and Additional Papers*, pages 256–268, 1987.

B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder, *CLU Reference Manual*, New York, NY: Springer-Verlag, 1981.

O. L. Madsen, Block Structure and Object Oriented Languages, *Research Directions in Object-Oriented Programming* (B. Shriver and P. Wegner, Eds.), pages 113–128, Cambridge, MA: MIT Press, 1987.

B. Meyer, *Object-Oriented Software Construction*, Hemel Hempstead, England: Prentice Hall, 1988.

B. Meyer, postings on Usenet (comp.lang.eiffel and comp.object), 1990.

M. Sakkinen, On the darker side of C++, *ECOOP '88 Proceedings* (S. Gjessing and K. Nygaard, Eds.), pages 162□176, Berlin and Heidelberg: Springer-Verlag, 1988.

M. Sakkinen, Disciplined inheritance, *ECOOP '89 Proceedings* (S. Cook, Ed.), pages 39□56, Cambridge, England: Cambridge University Press, 1989.

M. Sakkinen, Between classes and instances, aided by titles, manuscript, Jyväskylä, Finland: University of Jyväskylä, 1990.

M. Sakkinen, The darker side of C++ revisited, manuscript in preparation, 1991.

A. Snyder, Inheritance and the Development of Encapsulated Software Systems, *Research Directions in Object-Oriented Programming* (B. Shriver and P. Wegner, Eds.), pages 165□188, Cambridge, MA: MIT Press, 1987.

A. Snyder, Modeling the C++ Object Model: An Application of an Abstract Object Model, *ECOOP '91 Proceedings* (P. America, Ed.), pages 1□20, Berlin and Heidelberg: Springer-Verlag 1991.

B. Stroustrup, *The C++ Programming Language*, Reading, MA: Addison-Wesley, 1986.

B. Stroustrup, Multiple Inheritance for C++, *EUUG Spring '87 Conference Proceedings*, pages 189□207, 1987.

B. Stroustrup, Multiple Inheritance for C++, *Computing Systems*, 2(4): 367□395, Fall 1989.

B. Stroustrup, *The C++ Programming Language, Second Edition*, Reading, MA: Addison-Wesley, 1991a.

B. Stroustrup, private communication, 1991b.

D. Taenzer, M. Ganti, and S. Podar, Problems in Object-Oriented Software Reuse, *ECOOP '89 Proceedings* (S. Cook, Ed.), pages 25□38, Cambridge, England: Cambridge University Press, 1989.

J. Waldo, The Case For Multiple Inheritance in C++, *Computing Systems*, 4(2): 157-171, Spring 1991a.

J. Waldo, private communication, 1991b.

Corrigendum

I have afterwards noted some errors in the above paper. One of them is so significant that submitting a corrigendum seemed necessary; I am grateful to the editors (of Computing Systems) for agreeing to publish it. At the same time it is convenient to point out and correct the smaller mistakes. Most of these corrections had already been sent in, but by accident did no more get into the published version. John Skaller and other participants of the Usenet group 'comp.std.c++' must be acknowledged for pointing out one of the minor errors recently (see below); this lead me indirectly to discover the major one.

The significant change is that Restriction 2 in §5.5 — which I had felt as an unwelcome necessity — should be removed. Indeed, the argument about Example 12 that precedes the restriction is invalid, and the restriction would completely forbid inaccessible fork-join inheritance!

Think about a case in which the restriction is violated: a non-immediate ancestor `A` of class `C` is accessible to two intermediate classes `D` and `E` in the inheritance graph, but there is no class in the inheritance graph to which both `D` and `E` are accessible. The paths from `C` to `A` through `D` and `E` cannot then both be accessible, thus by the main Rule in §5.4 itself, they correspond to two disjoint `A` subobjects. Therefore it is fully feasible to have different redefinitions of `A`'s virtual functions in `D` and `E`: the restriction is not needed.

The reference given for the BETA language at the end of §3.5, [Madsen 1987], is not the most appropriate one. Another article from the same book should have been referred to: [Kristensen et al. 1987].

There is a slight misunderstanding about the accessibility of constructors of virtual base classes in §3.6. I supposed that the constructors of a virtual base class would be automatically visible to all descendants, ignoring access specifiers. This is not true, but instead, non-immediate descendants may need to declare that class also as a direct base class only in order to be able to invoke its constructor(s). A class may therefore be non-instantiable (abstract) also because it has no access to necessary base class constructors, and not only because of pure virtual functions [Skaller 1992].

It is an open question whether such an additional direct base declaration is always needed in current C++ in order to use explicit initialisers, even if the constructors of the indirect virtual base classes are *accessible* to the most derived class. On the one hand, it is said in Ellis & Stroustrup [1990 §12.6.2, p. 290]:

Initializers for immediate base classes [...] may be specified in the definition of a constructor.

Also in the example on p. 294 there is no other reason for such a redundant-looking declaration. On the other hand, the rationale given for the above rule is that multiple initialisations of the same base class subobject are prevented; but for virtual base classes this is already guaranteed by their special rules. Also, the HP compiler (Release 2.1) available to me did allow the constructor of a non-immediate virtual base class to be invoked.

There is a small but irritating clerical error in Example 7 (§4.2). Class `CowboyWindow` should be derived from `WWindow` and `CCowboy` instead of `Window` and `Cowboy`. Readers may have guessed that, because the example makes no sense otherwise.

The availability of the new book by Bertrand Meyer [1992] causes modifications to a couple of comparisons between C++ and Eiffel. Current Eiffel seems to allow, by renaming, the possibility that is desired in §4.2, in the second-last paragraph. It is called 'joining' in Meyer [1992 §10]. The types of the inherited routines (functions) that are joined in a subclass do not even need to be identical.

In §4.3, the last line of the second paragraph contains an incorrect comparison: inheriting a class without exporting any features in Eiffel corresponds to **protected** rather than **private** derivation in C++. The disadvantage of Eiffel described in the last paragraph has been corrected in the newest version of Eiffel. The following paragraph should be added to the end of §4.3.

So-called system-level validity checking in current Eiffel [Meyer 1992 §22] will detect this kind of attempted misuse of a FIXED_STACK object: not the assignment, but any invocation of an ARRAY routine. Thus Eiffel can now better than C++ enforce the protection of non-public features toward outside clients. However, Eiffel does not offer any protection toward descendant classes: there is nothing corresponding to **private** derivation (nor private members). The designer of class FIXED_STACK cannot therefore easily cancel the original decision to use ARRAY in the implementation, because some descendant classes may already depend on it. Also, Eiffel's system-level checking is pessimistic: it can reject even programs that would actually be type safe.

References

B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard, The BETA Programming Language, *Research Directions in Object-Oriented Programming* (B. Shriver and P. Wegner, Eds.), pages 7□48, Cambridge, MA: MIT Press, 1987.

B. Meyer, *Eiffel: the Language*, Hemel Hempstead, England: Prentice Hall, 1992.

J. Skaller, postings on Usenet (comp.std.c++), 1992.

CHAPTER 6

THE DARKER SIDE OF C++ REVISITED

To appear in Structured Programming Vol. 13 (1992), © Springer International 1992. — Published in parallel with the permission of Springer International. The journal version contains some editorial changes not found in this chapter, and the references are numeric. The reference given here as [Joyner 92] may be missing.

THE DARKER SIDE OF C++ REVISITED

Abstract

The C++ language is to a high degree a faithful follower of Simula as an object-oriented language geared more toward software engineering than exploratory programming. I highlight several of its noteworthy good ideas. Like Simula, C++ is designed to be a general-purpose procedural language and not ‘‘purely’’ object-oriented; this is not counted as a mortal sin here. However, taking the weakly typed and weakly structured language C as a base has become an irremediable handicap. Low-level concerns have caused at least one crucial flaw also in the object-oriented properties of C++. These basic defects have not changed since I first explored the darker side four years ago. Most, although not all, of the numerous features and facilities that have later been added to the language are obvious improvements. At the same time, they have increased the complexity of C++ so that it is now well comparable to Ada. (Some new features are suggested even here, nevertheless.) It is regrettable indeed if C++ becomes the *de facto* standard of object-oriented programming, but this danger looks imminent today.

1. Preliminaries

1.1. Introduction

The C++ programming language has become very popular during the last few years: there are many commercial implementations, new books of varying quality are coming out every month, conferences and journals are dedicated exclusively to C++, and an active discussion is going on in the Usenet newsgroups "comp.lang.c++" and "comp.std.c++" (the latter about the emerging standard). There is clearly a bandwagon effect, so common in the history of programming languages and of computing more generally. None the less is it relevant to examine what *intrinsic* merits and dismerits C++ has.

After my first critique of C++ [Sakkinen 88a] was written, the language has changed significantly. Release 2.0 of AT&T's C++ Translator, which brought many major changes as had been somewhat prematurely advertised in [Stroustrup 87a], was at long last released in June 1989. Even Release 2.1 in 1990 contained several new modifications. Release 2.1 is important because it has been defined more exactly than any earlier version of C++ [Ellis &c 90] and this definition has been accepted as the base document for the C++ standardisation committee founded by ANSI. Several other implementers beside AT&T (UNIX™ System Laboratories) now support largely the same features. Since C++ is a moving target, no article about it can claim to be absolutely up to date on all details, although the evolution will be slower than in the past. Release 3.0 is already being distributed from AT&T.

The paper [Sakkinen 88a] was to a large part based on personal experience with the C++ Translator, Releases 1.1 and 1.2 [Stroustrup 86, AT&T 85]. A while later, there appeared an article [Edelson &c 89] that takes into account some of the newer capabilities of C++ (e.g. multiple inheritance [Stroustrup 89b] is briefly mentioned), but clearly bases itself mostly on the published literature. It is a kind of follow-up to an earlier paper by the same authors on C [Pohl &c 88]. They have not been aware of [Sakkinen 88a], so these assessments are mutually independent. A more recent analysis, concerning exclusively the object-oriented aspects of C++, is [Snyder 91]. An extensive critique of C++ [Joyner 92] has been electronically published during the last revision of this paper. It was no more feasible to compare its findings and viewpoints to those of the earlier articles.

The discussion of C++ in the sequel is based on [Ellis &c 90] wherever no explicit reference is given. When it comes to really tricky combinations of details, probably even that remarkable manual cannot always give unambiguous answers, but the language remains operationally defined. I have some experience with Release 2.1, but not very much. Appendix A of [Hansen 90] was a practical checklist of the features that were new in Release 2.0. I have gained much more insight into object-oriented programming during the last few years, and therefore some judgements have changed from the old paper.

The evolution of C++, I am glad to acknowledge, has corrected several flaws pointed out in [Sakkinen 88a] and brought on substantial progress in many other areas. However, there have been also some small changes to the worse, in my opinion; and major new features have necessarily caused also some completely new problems. Further [Meyer 92 p. 500]:

> [...] the idea of orthogonality, popularized by Algol 68, does not live up to its promises: apparently unrelated aspects will produce strange combinations, which the language specification must cover explicitly.

The interactions between different aspects make it difficult also to write a paper on a programming language, if it is not focussed on one particular aspect. The current paper has in fact been in the making since the spring of 1990.

Two important new features will deliberately be left out of the discussion; they can be better assessed a year or two later. These are *templates*[1] (parameterised or generic classes) [Stroustrup 89a] and *exceptions* [Koenig &c 90; Sakkinen 91a]. Both are presented as ''experimental'' in [Ellis &c 90] and as fully existing features in [Stroustrup 91]. They were mentioned as future possibilities already in [Stroustrup 87b], which makes interesting reading in comparison to how the language has actually evolved during recent years. Templates are now implemented in C++ Release 3.0; their definition seems to be so macro-like that it causes a lot of problems [Cargill 92].

Some knowledge of object-oriented programming languages is a prerequisite for understanding this paper. It is also useful to know something about C++ or C, but I have seriously tried not to presuppose a thorough understanding — one of my purposes is to warn people about C++ and induce them to consider even other alternatives before committing themselves. Verbal argumentation will therefore be used much more than actual code examples; the latter would probably be more convenient to those readers who are already familiar with C++.

[1] The term 'template' seems to have a long tradition in this meaning: it is used already in [Conradi &c 74].

Unfortunately I cannot give a ready prescription what language to choose for the future. I would rather think that the prevailing object-oriented languages of 2000 are yet to be invented. However, people choosing a language for current needs should seriously evaluate at least Eiffel™ and one other language instead of automatically jumping on the C++ bandwagon.

I have not been able to grade all positive and negative items in this paper by importance, although some are clearly labelled as either very important or less important. Different aspects are significant for different people. This was recently illustrated by a long Usenet discussion about whether C++ should have an exponentiation operator. Very strong opinions were aired on both sides.

1.2. My angle of view

Before really starting to examine C++ itself, it is appropriate to state some of my basic beliefs and attitudes. Some of my prejudices are in fact favourable to the C++ approach, as opposed to Smalltalk™ [Goldberg &c 83] and its followers. C++ is a direct descendant of Simula [Dahl &c 68], which in turn antedated even the buzzword 'object oriented' by a good many years.

(1) I do not subscribe to the ''everything is an object'' philosophy. For instance, I do not like to regard integer *values* as first-class objects; integer *variables* are another story [MacLennan 82]. (2) I find it artificial that no function or procedure should be callable otherwise than as a "method" of some object[2]. (3) I do not like the requirement of a unique root class that is the common super class of all other classes — but even this requirement can be satisfied in a trivial way. (4) I dislike the reference semantics that most other object-oriented languages except C++ force on object variables.

One great divide in programming languages goes between "exploratory programming" languages that aim at great dynamism and run-time flexibility, and "software engineering" languages that have static typing and other features that aid verifiability and/or efficiency. While both kinds have their applications, I am more interested in the

[2] Of course it is always possible to ''repackage'' software that has ordinary (free-standing) procedures: define one class that contains them all as methods, or make a separate "wrapper" class around each single procedure.

latter group, to which C++ belongs. Smalltalk is the best-known representative of the former group. On points (1) through (3) above, it is Smalltalk that has deviated from the example of Simula; only on point (4) is C++ the more deviating language (although Simula objects can contain arrays, not only atomic components).

C++ in its current state is a rather large and complex language, and suggested additions are making it even more so. While excess complexity is certainly harmful, I suspect that some recent entrants in the camp of statically-typed object-oriented languages are already too Spartan to be convenient for general-purpose programming. I am in doubt about Modula-3 [Nelson &c 91], but rather sure about Oberon [Wirth 88], as partially explained in [Sakkinen 91b].

What is then unfavourable to C++ from the onset? Already in [Sakkinen 88a] I wrote:

> [...] when someone sets out to enrich an existing language with object-oriented or other *higher-level* features, trying to keep totally upward compatible with the base language can be problematic. Obviously, it is easier to extend a language that seems too restricted (e.g. Pascal) than one that has very general, powerful, and accordingly error-prone facilities (e.g. C).

Bertrand Meyer is more negative toward hybrid languages in [Meyer 89]:

> The search for compatibility at any cost is also the reason behind the centaurs sporting an object-oriented head on top of a C body, such as C++. Imagine this: on the one hand, inheritance, on the other hand, pointer arithmetic!

Of course, the creator of the Eiffel language has an axe to grind here, but I must mostly agree. Although C++ is in many ways a seamless whole, almost all its higher-level constructs and protections can be corrupted and circumvented at will by low-level manipulations (§2).

Today I take, with great conviction, a strong position on C extensions in general. The C language is so unsafe that striving to a total or almost total upward compatibility from C *cannot result in a good general-purpose object-oriented language.*[3] What can be had is an object-oriented language mostly suited for low-level systems programming. There certainly is need for such languages: they will boost productivity and quality where the current principal language is C or even assembler.

In the systems programming field Modula-3, mentioned above, looks like a strong contender. Two important features that Modula-3 supports but C++ does not are concurrency and garbage collection; but

[3] Therefore I cannot believe even Objective-C™ to be ''the solution'', although I know that it has evolved considerably from the version presented in [Cox 86].

unlike in most object-oriented languages, garbage collection can be used selectively in Modula-3 (including not at all if it is not desired). Another advantage of Modula-3 is that unsafe features, agreedly sometimes necessary for systems programming, can be isolated to a few unsafe modules.

In [Sakkinen 88a] I suggested the following as an advantage of C++:

> [...] previous C users can quite well upgrade *gradually* to programming in C++, in the first step just feeding their existing C code through the C++ translator and checking if some small modifications would be necessary.

Many people consider this rather a disadvantage. They claim that an abrupt change of paradigm is almost necessary to make programmers think in an object-oriented fashion. Therefore, it would be better to start with a language that *requires* object-oriented programming (e.g. Smalltalk or Eiffel), instead of one that merely *allows* it (e.g. Simula or C++).

Today I think that good object-oriented programming (OOP) is more a matter of restraint and moderation than of very powerful features and extremism. The distinctive properties of OOP seem to be rather tempting to be over used. The most distinctive feature of object orientation is certainly inheritance; anybody who has read some amount of recent OOP literature — not to mention object-oriented source code — must have encountered interesting misuses of inheritance.

1.3. An endemic culture

An often irritating feature in the writings of some developers of programming languages is that they contain very little references, especially to the work of researchers outside their own teams. This self-sufficiency is probably one reason why Oberon — as mentioned above — does not look to me on a par with the best object-oriented languages. The C++ community clearly suffers from this problem: you can easily see it when reading books and articles.

Other symptoms of self-sufficiency might be the following: The C++ literature uses some peculiar terminology, which can easily cause misunderstanding to other OOP people (cf. §3.1). Many seminars and tutorials even on topics like object-oriented *design* have the clause ''with C++'' in their names or advertisements, to assure potential attendees of staying safely in familiar territory.

The designers and supporters of some other object-oriented languages seem to have more interest in developments outside their own respective fraternities. They tend to advertise what they think to have done better than their contestants (including C++), and these advertisements may often appear over driven, not so seldom even containing some clear misunderstandings about other languages. But they also look out for things that are better in other languages, and think about similar improvements to their own.

It is naturally cosier for C++ researchers and practitioners to restrict their conversations largely to those people who like C++ and C. However, it is very useful and educative to listen to what other people, even staunch adversaries, have to say. Perhaps the object-oriented community as a whole is in some danger to remain or become too endemic, but it is so large and diversified that the danger is less severe than within a single language.

The above may be a bit exaggerated. In reality, e.g. the highly critical article [Sakkinen 92a] was warmly welcomed and quickly accepted to Computing Systems. It is to a large part a continuation of an active discussion about multiple inheritance [Cargill 91; Waldo 91]. The current paper, on the other hand, may too much serve to ‘‘convert the converted’’ in Structured Programming; maybe we would better need a paper on the averse side of Modula-2, Modula-3, or Oberon here.

2. C++ as a conventional language

2.1. Syntax

Let us now start the detailed examination with syntax, the most obvious facet of a programming language, although it is — especially in modern programming environments — among the least important facets. [4] Superficial syntactic differences may still divert a newcomer from the fact that C++ is an Algol-like language at the root. By 'syntax' we will understand also context-dependent syntax rules. The common usage of regarding all of them as ‘‘static semantics’’ and not part of syntax is well criticised in [Meek 90a], but drawing an exact line between syntax and semantics is a matter of taste.

On the lowest lexical level almost all current languages have regressed from Algol 60: the reserved words of each language are in no

[4] Pure syntax issues were not treated at all in [Sakkinen 88a].

way distinguished lexically (nor unambiguously by context) from programmer-defined identifiers but must compete in a common name space. This is especially harmful for the evolution and extension of existing languages: conflicts with new keywords can make previously correct code invalid. To avoid this, C++ has added remarkably few new reserved words into C, which in turn has aggravated the next problem.

I agree with the most common complaint about the syntax of C that it is overly terse. This shortcoming is listed in [Edelson &c 89], of course. In fact the paper claims C++ to be even worse than C, because

C, however, is also transparent; C++ is not.

It is probably meant that several things such as object initialisations happen implicitly ‘‘behind the programmer’s back’’ (§2.7). A remark that has often been seen elsewhere is that C++ stretches the syntactic framework of C near, if not beyond, the breaking point. For instance, the keyword ‘**static**’ has a lot of different meanings that must be distinguished by subtle differences in the context.

As [Edelson &c 89] notes, the type declaration syntax is confusing and error prone already in C, and becomes even more difficult with the additional type modifiers of C++. C++ also adds some confusion between the *tags* (allowed for compatibility with C) and *names* of structures, classes, unions, and enumerations. The antimathematical way to distinguish octal literals from decimal ones only by a leading zero was mentioned in [Pohl &c 88], and it persists in C++ too.

Like in Algol 60 and Pascal, in contrast to Algol 68 and Ada, the compound statement constructs of C and C++ have no end markers of their own, so programmers must remember to make blocks (i.e. add braces) at appropriate places. This is a minor nuisance; a minor convenience is that, unlike Algol and Pascal but like Ada and PL/1, the semicolon acts as a statement *terminator*, not a statement *separator*.

It is sometimes practical that C and C++ have a sequencing operator available within an expression, although it might have been more elegant to have an expression and not statement language in the first place, like Algol 68. The choice of comma for that operator is somewhat unfortunate because it can have a totally different meaning, in the argument list of a function call, depending on the number of enclosing parentheses. In array indexing it can be treacherous to people who are used to other languages: Suppose that ‘matrix’ is a two-dimensional array. An expression such as ‘matrix[j,k]’ is then fully legal, but it means row ‘k’ of ‘matrix’ (‘j’ is simply discarded). A single element must be denoted by an expression like ‘matrix[j][k]’.

The paper [Pohl &c 88] complains about the difficult visibility and scoping rules, and about the lack of function nesting as a separate item. The latter point is made also in [Edelson &c 89]. The additional constructs of C++ make visibility and scoping even more complicated than in

C. Prohibiting nested functions becomes a clear unorthogonality in C++, which it is not yet in C: see §3.2.

2.2. Conventional data types

As [Pohl &c 88] says, C has no Boolean or logical data type, nor has C++. The desirability of a distinct Boolean type in programming languages has been recently elaborated on in [Sakkinen 90] and [Meek 90b]. Among usual type constructors, there is no set constructor [Pohl &c 88]. In Ada there are no set types either, but small sets can be adequately and efficiently represented as Boolean arrays — this is not possible in C++.

All researchers of programming languages do not appreciate enumerated types, at least in object-oriented languages, but I certainly do if they are well defined [Sakkinen 91]. The approach to enumerations has fluctuated in C and, to a lesser extent, in C++. In [Pohl &c 88] bad inconsistency between different C implementations was found; [Edelson &c 89] notes that C++ and ANSI C defined enumerations to be identical to the **int** type and there was thus no more inconsistency. In my opinion [Sakkinen 88a], enumerations were completely superfluous under this definition, especially in C++ which already had a general means to define named constants.

Currently C++ regards every enumeration as a *distinct* type, and direct assignments between such types are not allowed. Unfortunately, enumerations are seen as *integral* types and thus any enumeration value can be automatically (i.e. without an explicit cast) converted to an integer. Arithmetic operators may therefore be freely applied to enumeration values, which does not make much sense.

As mentioned in the preceding subsection, the nesting of definitions within classes did not affect their scope in earlier versions of C++, but does in Release 2.1. This holds also for enumerations defined in a class, but they are accessible from outside the class by using explicit class qualification. These principles look very sensible.

The automatic promotion to **double** that made **float** into a kind of second-rate type was noted in [Pohl &c 88]; [Edelson &c 89] said that C++ and ANSI C define **float** into its own type, but this was actually implemented only in Release 2.0. I complained in [Sakkinen 88a]:

> For instance, **char** and **short** are something between full-fledged types and **int** crammed into a smaller space.

This was particularly harmful for function overloading. Now every different integral type, including the unsigned variants, is a first-rate type also with respect to overloading. As a related improvement, an explicit

signed char type has been introduced.

In current C++, the **const** and **volatile** declarators (modifiers) have subtle effects on types. As an example, for function overloading based on argument types, we have the type equalities **int** = **const int**, **int** = **int&** (reference type, see §2.5), **const int** = **const int&**, but **int&** ≠ **const int&** [Ellis &c 90 p. 307−308]. The reason is mainly that these modifiers do not affect values, only variables (objects).

Except for *bit fields*, which can be components of structures and are packed somehow in all implementations, C++ offers no means for a programmer to specify the alignment or packing of variables in storage, not even of the components of a structure or array. For low-level systems programming where software must sometimes adapt to queer hardware layouts, this is a surprising omission. Problems may arise also in interfacing to other programming languages, which is a relatively common need today. There is not even an equivalent of Pascal's '**packed**' attribute.

Union types can be useful even in an object-oriented language in my opinion, although not everybody agrees. Only *discriminated* (tagged) unions are acceptable from the viewpoint of type safety; also undiscriminated unions of *pointer* types if the type of the referent object is checked at run time. C and C++ unfortunately have undiscriminated unions only, and run-time type checking is not supported. The union types of C++ are thus extremely unsafe.

2.3. Array problems

I still maintain the opinion that the worst common feature of C and C++ is the handling of *arrays*, if the criterion is ''degree of badness weighted by importance'' [Sakkinen 88a]. Newer versions of C++ have not been and will not be able to bring any improvements, because they would break compatibility with older versions and C.

Array types are not first-rate types, either syntactically or semantically; they are subtly mixed up with pointer types. Once an array A has been defined, the name 'A' stands everywhere only for a pointer to its first element. There are no operations that treat an array as a whole [Sakkinen 88a]. One special case of this (mentioned also in [Pohl &c 88] and [Edelson &c 89]) is the paradox that an array cannot be returned as the value of a function as such, but can if it is wrapped in a one-element structure. Similarly, arrays cannot be passed by value as arguments to functions, unlike all other datatypes [Sakkinen 88a] (cf. §5.1).

It is difficult or impossible to implement array bounds checking, at least without making a lot of existing C and C++ code invalid

[Sakkinen 88a, Pohl &c 88] — see later paragraphs in this section. Strings are a special case that illustrate well the pitfalls [Sakkinen 88a,89a, Abrahams 88]. In both array and string operations, Fortran 77 is a more sophisticated language than C++.

The lack of true multi-dimensional and dynamic arrays is noted in [Pohl &c 88], however claiming that

> C is admirably more convenient for dealing with generic array processing than Pascal.

This holds only for ancient, ''Jensen and Wirth'' Pascal. If we consider ISO standard Pascal, rather the opposite is true: C and C++ have no feature comparable to its conformant arrays. Even more dynamic and versatile are the arrays of Algol 60 [Pohl &c 88] (''an improvement over most of its successors'', as Hoare said) and Ada, not to speak of APL.

It is true that, by virtue of the object-oriented extensibility of C++, a programmer can create as versatile array-like classes as he or she likes [Edelson &c 89]. Yet, even if such classes were available in a somewhat ''standard'' library, in many cases a programmer would face a dilemma: should I use the fancy array **class** or could I manage with the second-rate, built-in arrays — the latter will probably be more efficient and more compatible with existing code? The book [Ellis &c 90 p. 212] also concedes:

> [...] the C array concept is weak and beyond repair. The way to avoid this problem is to use a proper array object type such as the one presented in §14.2.

The section referred to presents a vector class using templates.

The article [Pohl &c 88] noted that initialisation was not possible for automatic arrays (and structures) in C. A similar problem in C++ was that one could not specify initialisers for an array of class objects [Sakkinen 88a]. These problems have been in part corrected in C++ Release 2.0, but it is still not possible to give initialisers for arrays created by the **new** operator.

Polymorphic or heterogeneous arrays are ones in which different elements may be of different types; all arrays are heterogeneous e.g. in Smalltalk. It is remarked in [Edelson &c 89] in one place that polymorphic arrays are not directly available in the C++ language or its standard libraries, but a programmer must spend time writing them if needed; in another place it is said that polymorphic arrays can be created using the inheritance mechanism or **void***. These seemingly conflicting statements are both true. Unlike ordinary arrays, polymorphic arrays can be constructed only out of pointers. Ordinarily one would define a class around them to achieve a clean interface and exactly the desired semantics. The general problems of polymorphic variables will be discussed in §3.6.

2.4. Drawbacks of pointer arithmetic

The rules of legal pointer arithmetic [Ellis &c 90 §5.7], and hence the rules of array subscripting, are such that enforcing them would be prohibitively expensive at run time. For that purpose, the run-time system would need to keep a directory of *all* objects of all types and storage classes. This seems inevitable because pointer values can be created even ''out of the blue'', e.g. by converting from integers[5]. Of course, the legality or illegality of *many* pointer-arithmetic and dereferencing operations could be inferred already at compile time.

Let A be an array of N elements of some type T. The legal pointer expressions derived from A then range from A to A+N (i.e. one element past the end of the array), but the result of trying to dereference A+N is undefined. Checks would therefore be needed *both* when pointer arithmetic is done *and* when a pointer is dereferenced — the worst of both worlds. For instance, every pointer should contain a ''past the end'' flag bit to signal the 'A+N' situation; this would cause a lot of overhead on typical machine architectures, where a hardware address takes a full word. The much-advertised efficiency of C and C++ would suffer badly from such checking; but that efficiency indeed comes mainly from neglecting safety.

To see the necessity of the ''past the end'' flag, suppose that another object B of the same type T as above happens to lie just after the last element of A at address A+N. The pointer expressions '&B' and 'A+N' will then correspond to the same machine address, but only the former may be dereferenced to access B. Additionally, the run-time object table should distinguish between one-element arrays with element type T, and non-array objects of type T. Incrementing a pointer of type T* by one is legal when it points to the former, and illegal when it points to the latter.

In my opinion, it is completely superfluous trickery in C and C++ that subscripting can be inverted, i.e. 'A[i]' is equal to 'i[A]'. It has been justified by the commutativity of pointer addition: both of the above are equal to '*(A+i)' and '*(i+A)'; but offering many different syntactic forms for one purpose can only make code less comprehensible. This is certainly a minor problem in comparison with the serious semantic defects mentioned above. However, it is aggravated in C++ by the fact that all

[5] A C++ implementation that did not allow any legal non-null pointer value to be obtained by conversion from any integer type, would not strictly break the rules of [Ellis &c 90 p. 67 − 68], but would be clearly contrary to the spirit, and would probably crash a lot of existing C++ software.

these forms can be overloaded independently of each other if A is not an ordinary array but a class object (§4.2).

The final irony comes from the following observation. The typical C idiom of traversing arrays by incrementing or decrementing pointers instead of subscripting was motivated by efficiency. Now it seems that on some new computer architectures, already this piece of code:

```
a[i] += b[i]; i++;
```

is not only easier to understand, but also more efficient than

```
*ap++ += *bp++;
```

2.5. Reference types and argument passing

Reference types [Ellis &c 90 §8.4.3], not found in C, are mainly syntactic sugar over constant pointers. The prime reason for their introduction must have been the requirements of overloaded operators (§4.3). It may also be difficult for programmers to remember the cases when they should pass the *address* of a variable as an actual argument to a function, instead of its *value*; references can help here.

This is best illustrated with an example. The function declaration − call pair,

```
void stuff1 (int *const p);
stuff1 (&number);
```

where '**int *const**' means that the formal argument is a constant pointer to integer, can be equivalently replaced by

```
void stuff2 (int& p);
stuff2 (number);
```

Thus we have essentially a second way to accomplish the same thing but with a slightly different syntax. This is against generally accepted language design principles.

A second use of a reference type is as the *return* type of a function; a function call can then appear on the left side of an assignment. Here, the syntactic advantage of replacing

```
int *const treat1 (int n);
*(treat1 (number)) += n;
```

with the equivalent

```
int& treat2 (int n);
treat2 (number) += n;
```

is a little greater than with a reference *argument*.

A third use of references is as aliases to variable names: after the definitions,

```
int x = 1234, &y = x;
```

'y' will be an alias for 'x'. This possibility is certainly more harmful than useful, but it is a necessary consequence of the definition of references. Note further that the syntax of reference declarations deviates from the logic of all other C and C++ declarations: the above indeed does not mean that the type of '&y' is **int**.

Reference types are clearly second-rate datatypes [Ellis &c 90 §8.4.3]. For instance, structures with reference components are allowed, but arrays of references are not. In many simple aliasing cases like the last example above, no actual reference variable need be allocated; but a formal argument or a structure component of a reference type does need storage. We cannot discuss all aspects of references here; they seem to add a lot of complexity to the language. C++ books have to explain their interactions with other language features in quite a number of places, and I suspect that many programmers still remain perplexed.

It would have been better to introduce references just as an argument-passing (and result-passing) mode as in most other languages. Indeed the array problems discussed in §2.3 could have been avoided if call by reference had been originally included in C. The C logic, which C++ has been more or less bound to follow, seems to have gone like this: (1) All arguments shall be passed by value; that is clean and simple. (2) We cannot afford to pass *arrays* by value; that is much too inefficient. (3) Let us invent a trick to reconcile (1) and (2): the name of an array shall not denote the array but only the address of its first element.

In fact, the whole principle of pass-by-value as it exists in many current programming languages is a remnant from the days of Algol 60 when only small entities such as integers were considered really values. That was the case also in C originally: structures were added to the language later, and arrays are not considered values even today. When the alternatives are variable and value arguments as in Pascal, semantic and pragmatic issues tend to conflict: to avoid costly copying, one often declares large formal arguments as variable even when they should by no means be modified within the procedure. A good set of alternatives would be: constant (**in** in Ada), variable (by reference, not exactly like **in out** in Ada), and possibly also result (**out** in Ada). Whether a constant argument is passed by value or by reference would then only be a pragmatic or implementation question.

2.6. Statements and expressions

A well-known misfeature in C and C++ escaped registering in all three previous papers, perhaps by being too obvious: it is completely legal to jump into any ‘‘structured programming’’ construct from the outside. A restriction in C++ is that one cannot jump into the scope of a variable, bypassing its initialisation. That prevents some unstructured jumps, but rather randomly. Relatedly, [Pohl &c 88] says that there are too many ways to jump *out of* a structured construct (or into another statement within it): **goto**, **break**, **continue**, **return**.

A defect in both **break** and **continue** is that they allow the programmer to exit or continue only the smallest enclosing loop or (**break** only) **switch** construct. No equivalent of Ada’s loop naming is available; on the other hand Ada has no equivalent to the **continue** statement. Especially badly designed is the **switch** statement. As [Pohl &c 88] remarks, every arm *must* be terminated by a **break** statement, or control falls through to the code for the following alternative. Even loop constructs may cross **switch** cases.

The **for** statement is criticised in [Pohl &c 88] for being too powerful and error prone. The power and generality has also advantages, of course. For instance, the very common need to traverse two data structures in parallel can be programmed nicely and symmetrically.

Both [Pohl &c 88] and [Edelson &c 89] regard the side-effect style of C and C++ as contrary to modern trends in programming languages. One factor in this style is that there is no assignment *statement*, although C and C++ are statement languages (§ 2.1). There are only various assignment *operators* whose chief purpose is their side effect, although their result value can also be used to build more complex expressions.

In addition to ordinary assignment there is, corresponding to almost every binary operator, a “modifying assignment operator” in Algol 68 style. As special cases there are additionally the unary increment and decrement operators, ‘++’ and ‘--’. In [Pohl &c 88] under the slogan, “Operator Set is Too Rich”, it is argued that all these operators (except ordinary assignment) may be superfluous because modern compilers can optimise conventional expressions and assignments to yield the same object code.

The effect of modifying assignment operators is *not* only optimisation: actually more important is that using them one can often avoid writing a complicated access expression twice, or creating temporary variables if the access expression has side effects. Therefore it is strange that there are no such operators formed from ordinary unary operators: the relative savings would be even greater. The increment and decrement operators in turn have no corresponding ordinary operators (‘succ’ and 'pred' in Pascal).

The example of Mode [Vihavainen 87] shows that at least the same benefits can be obtained with only *one* additional operator (or actually pseudo-variable) as with a large number of assignment operators. In an assignment in Mode, the symbol '*' may be freely used on the right-hand side to refer to the entity denoted by the left-hand side (its old value). This facility is indeed much more general than the operators of Algol 68, C, and C++. As an example:

```
bucket.value (element) := */n + 1 - p/*;
```

Both [Sakkinen 88a], [Pohl &c 88] and [Edelson &c 89] complain that the order of evaluation of subexpressions is too implementation dependent. At least some improvement has happened in Release 2.0; we read in [Ellis &c 90] that

> ... the usual mathematical rules for associativity and commutativity of operators may be applied only where the operators really are associative and commutative.

Incidentally, the excuse given in [Edelson &c 89] for the old state of affairs:

> C++ could not rigorously define subexpression evaluation order without diminishing compatibility with existing code.

is misleading: changing a language definition in the *opposite* direction would break compatibility.

2.7. Miscellaneous

The PL/1 language, initiated by IBM in the mid-sixties, was not nearly as bad as current folklore portrays it; mainly too ambitious and all-embracing for its time. One of the lessons that most later language designers learnt from PL/1 (and Algol 68) is that very liberal automatic type conversions, designed for the convenience of programmers, weaken static typing. They tend to cause a lot of harm by concealing programming errors. This lesson has been largely ignored in C++: it offers several automatic conversions, some of which can be user-defined (§5.1).

Explicit type conversions or *casts* are on one hand less dangerous than implicit conversions because they can be noted in the source code. They are probably needed much less often in C++ than in C [Stroustrup 91 §3.2.5], so they may be scarce enough in good C++ source code to remain conspicuous even when the ''functional'' notation recommended by Stroustrup is used. On the other hand they are more dangerous because they can be used for punning just as well as for semantically sensible conversions. Punning means that the same bit pattern is merely interpreted as if the value were of another type than it originally was; typical examples are conversions between pointers and integral types. A more subtle example of the dangers of explicit type conversion will be given in §3.3.

Taking the above points and §2.2 through §2.4 into account, we can fairly say that C++ is a *weakly typed* language if Pascal is strongly typed. We can similarly, based on the observations of §2.6, classify it as *weakly structured* if Pascal is structured. There is one aspect on which C++ is more structured than Pascal and many other languages: non-local jumps (i.e. from one function to another) are not possible. However, even Pascal allows such jumps only from a nested procedure or function to a containing one, a situation that does not occur in C++ except in the rare case of a local class (§3.2).

In many cases, C++ causes the automatic creation of temporary objects of which the programmer may not easily be aware. The use of temporaries is indeed left to the discretion of implementers [Ellis &c 90 §12.2] — very good implementations will probably introduce temporary objects in fewer situations than mediocre ones. With traditional datatypes (pure values) this is only a pragmatic matter that should not bother the programmer at all. On the opposite, with complex and sophisticated classes the creation of a new instance may well have semantic side effects that cannot be completely undone when the instance is deleted.

The *preprocessor* facility of C++ is inherited from C, and feels like a relict from the sixties. Preprocessing directives are rather foreign to the language proper, and they operate only on the lexical level. Compared to any reasonable macro assembler, they are rudimentary. Fortunately, there is less need to use preprocessor directives in C++ than in C (especially pre-ANSI C). The designers of almost all new high-level languages not based on C have omitted facilities like macros or conditional compilation, although they were more fashionable still in the seventies[6]. Such features can sometimes be very practical, but they can be obtained by using a free-standing macro processor.

The preprocessor directive '**#include**' is the only means in C++ for defining export-import relationships between modules. It is a poor substitute for the facilities of most other object-oriented languages or e.g. Modula-2. Even the modest Dee language [Grogono 91] has a very nice modularisation, where there is one *canonical document* for each class, and interface files needed to transfer information to the descendants and

[6] "All programming languages (not only assembly languages!) should support macroes." [Conradi &c 74].

clients of the class are automatically taken care of. — For instance, it is only a convention in C++ that class declarations and similar items are usually grouped into separate "header" or "interface" files. Although the language itself has no support for orderly modularisation, today there are many comprehensive programming environments that can help. This problem is therefore lessening in importance.

On the other hand, the typical OOPL practice that only a *single* class can be a unit of source code modularisation is often a disadvantage, e.g. when there are several classes with intimate interdependences. In this respect, C++ (like Modula-3, Oberon, BETA, etc.) is more convenient.

3. Classes and objects

3.1. Terminology

A few terms that C++ literature uses about class-related things can in my opinion be somewhat misleading, and certainly conflicting with "main-stream" object-oriented terms; this applies a little less to [Ellis &c 90] and other newer books and papers than [Stroustrup 86]. Therefore, I will use the word 'class' in the conventional meaning, comprising mainly: a template for creating objects, a collection of functions and common data, and, in C++ only conceptually, also the set of its existing instances. The words 'object' and 'instance' (instead of 'class' or 'class object') will be used to mean 'an instance of a class'.

I especially dislike the usage of the word 'member' — a synonym of 'element' in set theory — in the meaning of 'component'. Because the word 'static' is so overloaded (§2.1), I will use 'common' instead in the specific meaning of 'shared among all instances of a class'. When needed, I will use 'instance' as an adjective meaning the opposite of 'common'.

The new meaning of 'virtual' introduced with multiple inheritance [Sakkinen 92a] is also confusingly different from what 'virtual function' means, so I will use the distinct terms, 'sharable' for 'virtual' and 'duplicatable' for 'non-virtual'. *Shared* or *duplicated* subobjects (§3.8) of a class C appear only in "fork-join inheritance" [Sakkinen 89b], in which a non-immediate subclass inherits C over more than one path.

The use of 'derived class' and 'base class' instead of 'subclass' and 'superclass' does not sound too unnatural to me; both pairs of terms will be used, as well as 'descendant' and 'ancestor'. These relationships will not be supposed to be *direct* or *immediate* unless explicitly so stated.

In my opinion, some other terms used in connexion with C++ are more appropriate than their equivalents favoured by the Smalltalk community. For instance, it is misleading to speak about ''message passing'' instead of 'late binding' in Smalltalk. According to the normal practice of scientific nomenclature, concepts originating from Simula should retain their original names. Such reserved words in C++ as 'virtual' (in the original meaning: §3.4) and 'this' (instead of 'self') are faithful to the Simula tradition.

We get the following translation table (Table 1):

Table 1. Some different terms for similar meanings.
'[...]' denotes an optional part

C++ literature	Smalltalk literature	This paper
class	class	class
class [object]	instance	[class] object, instance
[non-static] data member	instance variable	[instance] data component
[non-static] function member	instance method	[instance] function component, insta
static data member	class variable	common data component
static function member	class method	common function [component]
base class	superclass	*both*, ancestor
derived class	subclass	*both*, descendant
virtual (base class)	—	sharable
non-virtual (base class)	—	duplicatable
this	self	this

In C++ literature, 'member function' actually appears more frequently than 'function member'.

Common functions are a novelty of Release 2.0. Their main difference from instance functions is that they need not be invoked as components of any object. Therefore the variable **this**, which is automatically defined for every instance function to point to the object for which the function is called (the 'receiver' object in Smalltalk parlance), is not available to common functions. They cannot be virtual, either; that appears reasonable although not inevitable.

3.2. Class declarations (definitions)

C++ class declarations need not follow the structure of e.g. Smalltalk, where the main division is: common functions, common data components, instance functions, instance data components. Fortunately, a programmer *can* use this grouping to make the structure of each instance and of the class's common data clearly visible; every common component must just be separately declared **static**. A class declaration can contain still other things beyond those four categories: type definitions (including nested class definitions) and **friend** declarations. Normally, the actual *definition* (code) of each function is somewhere outside the class definition; if it is inside, the function will automatically be treated as **inline**.

The supported structuring of a class declaration is by access levels: **public**, **protected**, and **private**; there can be more than one section of the same level. This is also a sensible division, because it stresses the different interfaces of the class. The friend declarations are the only ones that do not fit well into this division, because the access levels do no affect them. When multiple inheritance was added to C++, it would have been consistent to impose the same division on the list of immediate superclasses. Instead, the access level must be given for each superclass individually ('private' being the default).

Already in C, a structure can have components of type pointer-to-function. This is one major factor in the suitability of C for an intermediate language in the implementation of OOPLs. In C++, such components are *data* parts, but the difference between them and function components is subtle. One of the implicit conversions of C++ even allows the dereferencing operator to be elided in invocations through a function pointer, so there need not be any syntactic difference between the call of a function component and a call through a function pointer component.

C++ allows class definitions to be nested, but nesting originally had no significance for scoping or visibility — a promising source of confusion. This has been changed in Release 2.1 so that scopes nest in the natural manner [Ellis &c 90 §9.7]; the incompatibility does not cause much harm, because hardly anybody would have nested class definitions earlier. It would have been a much more elegant choice from the beginning to allow a general nesting of all constructs (including functions) with consistent scope rules, i.e. to follow the example of Simula.

The unorthogonality mentioned in §2.1 comes about as follows. Functions consist of blocks, blocks *may* be nested and types defined within them (already in C). A class defined within a block is called a *local class*. Such a class definition can contain function definitions, not only of function components but also of friend functions. Thus, a function definition can be indirectly nested within another one; but it has no access to the automatic (stack-allocated) variables of the enclosing function [Ellis &c 90 §9.8].

A local class must be an exceptional case in real programming. For instance, all functions of the class must then be written completely within the class definition, because there is no other place where the class is visible *and* function definitions are allowed. There is no such constraint on a class declaration nested within another class, because multiple class qualifications such as 'Outer::Inner::some_function()' can be used to denote components of nested classes.

3.3. Inheritance

I have treated the inheritance principles of C++ in great detail in [Sakkinen 92a, 92b]. Contrarily to my previous positive view [Sakkinen 89b], I now found several rules that should be amended, especially to make multiple inheritance semantically more consistent. The prime suggestion is that the sharing or duplication of multiply inherited super classes should be implied by the inheritance modes (public, protected, or private), instead of being declared explicitly and independently. The modifications look rather harmless to other parts of the language, but it is very unlikely that the C++ community (most importantly the ANSI committee) would adopt any of them.

With the current C++ rules, one can easily construct classes with perhaps more obvious anomalies still than in the examples of [Sakkinen 92a]. For instance, let us define:

```
class A { ... };
class B: public virtual A { ... };
class C: public B { ... };
class D: public B { ... };
class E: public B, public C, public D { ... };
```

Every E object will then have three distinct B subobjects, but these will share a common A subobject. This is badly inconsistent with the notion of B being a subclass of A.

It is worth acknowledging even here that private inheritance is a semantically and conceptually valuable capability, although it also makes the language more complex. It has been a feature of C++ from the beginning, but very few other languages support anything similar, although the possibility of private *components* is quite common. Dee [Grogono 91] is one language that has followed the example but not completely: 'inherits' corresponds to public, 'extends' to protected (not private) inheritance.

One observation about the access levels of class components and superclasses is missing even fr om [Sakkinen 92a]. Already in early ver - sions of C++ wher e only the access levels 'private' and 'public' existed, access to private components (and superclasses) could be granted to some other class or function by a **friend** declaration. When the 'protected' access level was added, no analogy of 'friend' ('comrade'?) was intr o- duced for allowing access to protected components. After the access rules of protected components wer e strengthened [Sakkinen 92a §3.1], a class could need to declare even its own subclasses additionally as comrades!

The above proposal would fit well with one suggestion made for the Law of Demeter [Sakkinen 88b]: an explicit 'acquaintance' declaration would be r equired for a class or function to access even public compo- nents of another class, unless it has access rights accor ding to the basic Law of Demeter [Lieber herr &c 88]. (This was a r elaxation of the Law , although it would be a restriction if added to C++ as such.)

A pointer to a subclass can be implicitly converted to a pointer to a superclass under proper conditions, for instance when the inheritance is public and the super class is not duplicated. On the other hand, arbitrary conversions between pointer types ar e possible by explicit casts. Obvi- ously, an explicit conversion is again more dangerous than an implicit one (§2.7). There could be a need for ' 'safe casts" in some circumstances. Consider this example:

```
class A { ... };
class B { ... };
class C { ... };
void do_something (A*);
void do_something (B*);
void do_something (C*);
class D : public A, public B { ... };
D *pointer;
```

If we want to invoke one version of the overloaded function 'do_some- thing' on 'pointer', an explicit cast to type A* or B* is needed because the call would otherwise be ambiguous. However, using a cast allows us just as well to do the following:

```
do_something ((C*)pointer);
```

which would typically lead to disaster at run time.

3.4. Virtual functions

An instance function can be declared **virtual** in C++ with the same meaning as in Simula. If a virtual function is redefined in any subclass, its invocations will be bound at run time to the appropriate version of the function, based on the actual class of the object as whose component the function is invoked — *late binding*. In Smalltalk and many other OOPLs, all operations are automatically late-bound; this is sometimes even put forward as a requirement of true object orientation, but I do not agree.

C++ has the essential difference to Simula that late binding can be prevented by class qualification ('Myclass::clobber(thing)'). This feature allows e.g. a function redefinition to call the superclass function to be overridden (as by using 'super' in Smalltalk), and thus corrects an obvious defect of Simula. Unfortunately, it can also be misused. It is written in [Ellis &c 90 p. 210]:

> As a rule of thumb, explicit qualification should be used only to access base class members from a member of a derived class.

I consider it sometimes fully reasonable also to early-bind virtual functions of the invoking class itself [Sakkinen 89b]. The questionable case is explicit qualification used by *external* clients; even that is allowed by the language, although discouraged in the above quote.

A useful new facility for classes in Release 2.0 are *pure virtual* component functions — equivalent to *deferred* routines in Eiffel [Meyer 88]. Formerly, unlike Simula, C++ required every virtual function to be defined (i.e. code and not only the function header to be written) in the first class where it was declared. Now it can be left explicitly undefined there, syntactically by adding '= 0' after the function header; this makes sense if the name of the function is considered to denote a pointer to the actual function. Any class that has some pure virtual function component is considered an *abstract class*: no direct instances of it can be created, it is meant only to serve as a superclass for inheritance. Those subclasses (not necessarily immediate) that provide code for all inherited pure virtual functions are ordinary, *concrete* classes. It is in general better to inherit from an abstract than from a concrete class [Johnson &c 88].

Actually, it is possible to define a function even in a class in which it is declared as pure virtual, but it can then be invoked only by using explicit class qualification. This possibility looks like an unnecessary complication to me. It would also be better if a *class* were to be declared abstract in the first place, and only after that could some functions be pure virtual, like in Eiffel.

There seems to be currently some confusion about pure virtual functions in multiple inheritance, exhibited by some compilers [Skaller 92]: Suppose that classes B and C inherit class A, and D further inherits both B and C (a simple fork-join structure). If a pure virtual function of A is defined in B but not in C nor D, then those compilers make the function pure virtual again in D, causing D to become an abstract class. This conflicts with the dominance rule for virtual function definitions. The reason of this surprising behaviour is said to be one single sentence in [Ellis &c 90 §10.3]:

> Pure virtual functions are inherited as pure virtual functions.

Reading the whole context, it is evident to me that the implications of this sentence have been interpreted erroneously.

3.5. The fundamental defect: type loss

I wrote in [Sakkinen 88a §5]:

> Classes in C++ are defined in such a way that a **struct** becomes just a special case of a **class**, which is nice economy of concept.

In contrast, [Edelson &c 89] says:

> Having both **struct** and **class** is redundant. A **struct** is the same as a **class** that has by default **public** components. The **struct** keyword could not be eliminated without losing compatibility, but the **class** keyword is unnecessary.

Today I would agree more with the latter view, but both quotes miss one essential point. I venture to claim that herein lies *the fundamental defect that renders C++ insufficiently object-oriented,* even disregarding C-level tricks (cf. §1.2). Structures could have been left as they are in C, and classes should not be mere generalisations of structure types. This was suggested in [Sakkinen 88a §9], but it is worth restating a little more thoroughly.

Every object should have at its beginning some kind of *descriptor,* as is the case in practically all object-oriented languages ever since Simula. The descriptor should at a minimum indicate the class of the object. It would help maintain the *identity* and *integrity* of the object. Obviously, there should also exist some kind of run-time descriptor for each class. This does not imply that classes should be first-rate objects themselves (except possibly constant objects); that would not be desirable in a compiled, statically typed language.

One reason for omitting object descriptors must have been an overemphasis on ''efficiency''. Indeed, in degenerate cases C++ gives no time or space penalty for declaring and using a class instead of a C structure. The situation changes when a class has at least one **virtual** function. There must be a run-time *virtual function table* (or vector) corresponding to the class, and each instance of the class must contain a pointer to it, thus a kind of descriptor. Unfortunately, at least in those older implementations that I am familiar with, this descriptor is not at the beginning of the object. If you have an untyped pointer (**void***) to an object in C++, you therefore cannot get any information about its class or size.

The fundamental defect is often called '*type loss*' for obvious reasons: any knowledge about the type of an object that we do not maintain statically cannot be safely recovered. Type loss happens whenever a pointer to a subclasss is assigned to a pointer to a super class (see example in §3.8). We will call that a 'type loss of the first kind'. Another kind occurs when an *object* of a subclass is assigned to an object of a super class (§5.1). Type loss of the second kind is absolutely irrecoverable; Lea's suggestion (§3.7) would prevent it.

It is admitted in [Ellis &c 90 p. 212−213] that the lack of run-time type information makes some programming tasks difficult. Two reasons are given for not including such information in objects. First:

> This requirement would compromise object layout compatibility with languages such as C and Fortran, which is too high a price to pay [...].

Low-level considerations seem to take precedence over object orientation here[7]. If objects were distinguished from structures as suggested above, this argument would not count: it would be sufficient for structures to be compatible with other languages. Such compatibility may often be impossible anyway, because C and C++ do not allow the programmer to specify the alignment and packing of structure components (§2.2).

Second:

> ... would enable a style of programming that relies on switching on a type field rather than using virtual functions. [...] in Simula programs this style has lead to messy, non modular code ...

This in turn is the attitude of which the C and C++ community often accuses more strongly typed and structured languages: ''the programmer must not be trusted''[8]. Inheritance and virtual functions are not a panacea, and they can be misused as well. I will elaborate further on this point in the sequel.

---

[7] Incidentally, Fortran has not even had any record types before the new Fortran90 standard.

[8] The very good index of [Stroustrup 91] has only two subentries under 'misuse': 'of C++' and 'of run-time type information'!

3.6. Polymorphic variables and "typecase" programming

Variables that can only hold values of exactly one type are often called *monomorphic*, and those that can at different times hold values of different types are called *polymorphic*. I would like to further divide the latter into *freely* and *restrictedly* polymorphic variables. In languages without static typing, whether object-oriented (Smalltalk) or not (LISP), all variables are freely polymorphic. In statically typed non-OO languages, all variables are monomorphic[9]. It is conventional and useful to distinguish between the *static* (declared) type and the *dynamic* (run-time) type of a polymorphic variable.

In statically typed OO languages, usually all variables are restrictedly polymorphic in a special way which is called *inheritance polymorphism*: a variable can hold an object of either its declared class or any subclass of that. Freely polymorphic variables can be declared in many such languages as a special case of inheritance polymorphism because there is a unique class (e.g. 'Object') that is a superclass of all other classes. Simula has a special untyped (universal) pointer type, in addition to statically typed pointers, which are inheritance-polymorphic.

In C++, only pointers and references to class objects are inheritance-polymorphic, all other variables are monomorphic — but see §3.7 for a suggested language extension. Additionally, there exists the universal (freely polymorphic) pointer type **void***. This corresponds to untyped pointers in Simula, but **void*** is not restricted to point to class instances; remember that not all datatypes in Simula and C++ are classes.

Freely polymorphic pointers are necessary in C++ in the linkage to such low-level routines as memory allocators and deallocators. Otherwise they are almost useless due to the ''fundamental defect'': if we only know the address of an object and nothing about its type or even size, we can hardly do anything sensible with it. Exceptional cases are those in which the type of the object is somehow represented somewhere else, e.g. in the argument list of the 'scanf' function from the C standard library[10]. Such usage is rather the antithesis of object orientation, however.

Inheritance-polymorphic pointers are, of course, those that are most often required. All features of the static class of such a pointer can be used without violating type safety, and late binding takes care of the dynamic class of the referenced object. There are, however, situations in

[9] Except for unions: see §2.2

[10] Standard C input and output functions are ''officially'' callable from C++, although the genuinely C++ *streams* library is preferred.

which *freely* polymorphic variables are useful in languages with run-time type checking and type inquiry. Such situations cannot be handled naturally in C++ (but see [Stroustrup 91 §3.5]: Run-time Type Information). It is symptomatic that large general-purpose class libraries for C++ mostly have built an additional, ''more object-oriented'' layer of their own. They typically have one root class from which almost all other classes in the library are derived, and type inquiry is supported in that hierarchy.

When talking about arrays, sets, and other collections or containers, the customary terms are 'heterogeneous' and 'homogeneous' instead of 'polymorphic' and 'monomorphic', respectively. It should be noted that genericity or parametrised classes (*templates* in C++) mainly help programmers to declare open-ended sets of *homogeneous* collection classes; they do not give any new facilities for coping with heterogeneity.

I think that there is a conceptual flaw in Stroustrup's aversion to "typecase programming" (§3.5). The following simple inheritance-polymorphic example should serve as an illustration.

```
class Animal { ... };  // abstract class
class Fish: public Animal { ... };
class Bird: public Animal { ... };
class Mammal: public Animal { ... };
class Nature_watcher { ...
    virtual observe (Animal *target);
    ... };
class Cook { ...
    virtual prepare_meal (Animal *ingredient);
    ... };
```

Both 'observe' and 'prepare_meal' would probably be highly dependent on the dynamic class of their argument, but there is no natural way to take it into account in C++.

If C++ supported multiple dispatching (§4.1), it would be possible to make the late binding of the above functions depend on the classes of both the ''owner'' object and the argument. The functions would thus be virtual in both Animal, Nature_watcher, and Cook. But it is not intrinsic to animals how somebody observes them or makes food out of them. To avoid excessive coupling between classes, the designer of the Animal class should not need to know anything about the classes Nature_watcher or Cook.

The only reasonable way to implement the above example would be by typecase programming. Judiciously used, it can be advantageous for code management as well, in comparison to multiple dispatching: The number of different functions to be maintained is smaller if one does not define a different virtual 'observe' for each subclass of Animal. If most of the code of 'observe' is common for all subclasses, the solution by virtual functions would also cause a lot of code duplication — a problem that object-oriented programming is supposed to prevent.

3.7. Storage classes and garbage collection

The great majority of object-oriented pr ogramming languages, as well as LISP and CLU [Liskov &c 81], ar e based on *reference semantics*: variables cannot contain objects (except some atomic values) but only pointers to them. This means in practice that all objects ar e allocated on the heap. C++ is among the exceptions that support *value semantics*: objects may belong to any storage class (in the C sense) and may directly contain other objects. I regard this as an *advantage* of C++, and at least it makes C++ clearly more homogeneous than many other object-oriented extensions of conventional languages, e.g. Simula and Objective-C [Cox 86]. BETA [Kristensen &c 87] is another established language that allows value semantics. In recent versions of Eif fel, one can define ‘ ‘expanded types’’ [Meyer 92 §12.2] for such behaviour.

A consequence of r eference semantics in the other languages is that automatic garbage collection (GC) is both highly desirable and rather easy to implement. It is noted as a disadvantage of C++ in [Edelson &c 89] that ther e is no GC but programmers must take car e to explicitly delete objects that ar e no more needed. However, this problem does not concern at all those objects that ar e allocated on the stack or statically; the *need* for garbage collection is not so gr eat in C++ as in Smalltalk, LISP , or CLU. The danger of *dangling pointers* in C and C++ is, on the other hand, even greater than in Pascal: pointers are not limited to point to heap-allocated objects only , they can point to parts in the middle of allocated objects, and above all there is pointer arithmetic.

The article [Boehm &c 88] r eports on an appr oach that succeeded remarkably well in adding GC to existing C (not even C++) softwar e by replacing the standard dynamic memory allocator with a garbage-collecting one. However, the success was in part due to the fact that the code in question made no clever tricks with pointers, such as:

```
Person *p = new Person;
long j = long(p) / 100;
long k = long(p) % 100;
p = new Person;
        // Any garbage collector would now regard
        // the first Person object as unreachable.
Person *q = (Person*) (100 * j + k);  // But here we get at it again!
```

Very obviously, completely safe garbage collection cannot be added to *full* C or C++, i.e. without r estricting pointer operations. This holds even for so-called *conservative* GC.

It is suggested in [Edelson &c 89]:

> Given the fact that garbage collection is not in the language it should be possible to design a **class** which causes instances of **classes** derived from it to be garbage collected. Such a **class** if implemented robustly and distributed in a standard library could be quite useful.

Unfortunately, even this is not so straightforward as might appear from the quote, because the designer of a C++ class cannot control where and how *pointers* to instances of the class are manipulated[11]. Both the approach of [Bartlett 89] and that of [Edelson 90] need several rules that must be obeyed in order to make specific classes garbage-collectable. If a class should both be amenable to these *copying* GC methods and have a *destructor* (§5.2), additional tricks must be defined to get the destructor invoked [Wachowitz 91].

One less desirable consequence of the C++ approach is that an object-valued variable (in contrast to a pointer-valued one) is always monomorphic, as mentioned in the previous subsection. This is contrary to what people expect from object-oriented programming, and means that one is after all forced to use pointer variables or separate ''handle classes'' [Stroustrup 91 §13.9], and heap allocation of the actual objects if one wants to exploit e.g. late binding.

The paper [Lea 90] points out the above problem and suggests the addition of inheritance-polymorphic object-valued variables to the language to solve it, concretely by overloading the keyword '**template**'. The suggestion could be implemented by variables with value semantics but "pointer pragmatics": essentially, the compiler would automatically create handle classes and programmers would not have to care about them. Reciprocally, Lea proposes that it should be possible by declare monomorphic pointers and and references by using the new keyword '**exact**'. Both ideas look sensible.

There is yet another point in favour of C++. Most other object-oriented languages do not offer any explicit *deletion* operation for objects (§5.2); the removal of unneeded object happens *only* by means of garbage collection. Since objects are *created* explicitly, this is asymmetric (conversely, in C++ some object creations are rather implicit). Indeed one can think that all created objects conceptually live for ever; but at least for modelling many real-world objects, a meaningful destruction at a well-defined point in time would be desired. Garbage collection is in a sense better suited to value-oriented than object-oriented languages.

[11] By overloading the address-of operator — although I am opposing that possibility in §4.2 — and declaring it private, some degree of control is possible.

3.8. Object identity

The crucial concept of object *identity* [Khoshafian &c 86] is not as strong in C++ as it could and should be. The language knows only addresses (pointers and references), with some automatic adjustments necessitated by multiple inheritance. To see the problems, we start by dividing objects into three categories: complete (free-standing) objects, superclass subobjects, and component subobjects. In the more typical OOPLs with reference semantics only, there are no truly contained component subobjects: instance variables are references to other complete objects.

The paper [Snyder 91] regards component subobjects as objects with identities of their own even in C++, and this is certainly the right choice. If we have a reference to an object O, we have no possibility to know whether O is a component subobject or not. If we have also a reference to another object P, we *can* check whether O is a component subobject of P or not, but only by writing code specific to the class of P. However, most other OOPLs are no better than C++ in this respect. In object-oriented database systems it is more common to have support for *composite objects*, such that it is possible to get from the parts to the whole. A good example is ORION [Kim &c 89].

As for superclass subobjects, Snyder presents two alternatives, the multi-object model and the monolithic object model. He writes:

> We prefer the monolithic object model, both because it is simpler, and because it is more consistent with the "mainstream" concept of object.

In most other languages, superclass subobjects really have no identities of their own and cannot be addressed. For instance, the pseudo-variable 'super' in Smalltalk refers to the same object as 'self'; it only implies a specific class for the method search.

Snyder admits that the monolithic model cannot account for those C++ inheritance structures in which an object visibly contains several subobjects of the same superclass. The rules proposed in [Sakkinen 92] would indeed prevent such class hierarchies. Unfortunately, the monolithic object model is still incorrect under that restriction because a superclass subobject in C++ does have an identity of its own, almost like a component subobject. As an example:

```
class A { ... };
class B { ... };
class C : public Z, public A { ...
    B part
    ... };
C *Cpointer;  B *Bpointer;  A *Apointer;
```

The variable Cpointer is thus of type 'pointer to C', and so on. When

Cpointer is referring to an instance of C, we can get the address of its 'part' (subobject) component by

```
Bpointer = &(Cpointer->part);
```

and the address of its A (superclass) component by

```
Apointer = Cpointer;
```

Note that there is an automatic conversion in the latter assignment; no explicit cast is needed.

For the monolithic object model to be adequate, there should also be a way to get from an A* to a C*. At first sight, there seems to be a way through a "reverse" pointer cast:

```
Cpointer = (C*)Apointer;
```

Unfortunately, this cast is extremely unsafe due to the ''fundamental defect'': it cannot be checked whether the A object that Apointer refers to is really a subobject of a C object or not. (The equivalent operation is checked at run time in Simula and Eiffel.) Further, such a cast is totally disallowed (for implementation reasons) if A is a *sharable* (virtual) superclass of C [Ellis &c 90 §10.6c]. The monolithic model actually requires all superclasses to be sharable!

The situation is essentially similar as with component subobjects: If we have a pointer to an object O of class A, we have no possibility to know whether O is a superclass subobject or not. If we have also a pointer to another object P of class C, we *can* check whether O is a superclass subobject of P or not, but only by writing code specific to C. Only if both pointers are statically typed, C* and A* and not of the generic pointer type **void***, *and* if there is only one superclass component of type A in P, will simple pointer comparison suffice (with an explicit type cast possibly required).

4. Overloading

4.1. Overloading versus multiple dispatching

As in most literature, by *overloading* we mean that several entities in the same scope can have the same name, and the correct entity is selected at compilation time based on the context where the name occurs. In C++, functions can be overloaded if they have different argument signatures.

The main principle of function overloading in C++ is very sound. First, different return types are not yet a sufficient difference. Otherwise the determination of the type yielded by a function invocation would be problematic; with this rule, the determination of the type stays strictly bottom-up. Second, invocations with type combinations of actual arguments such that no unique overloaded function matches *all* argument types best, are rejected as ambiguous at compile time [Ellis &c 90 §13.2]. This conforms better to the spirit of software engineering than would the arbitrary choice of one among several "best" alternatives.

One difficulty with overloading is that programmers can easily confuse it with late binding (virtual functions) in some situations. If an instance function of a class C is redefined in a subclass D, the difference between a virtual and non-virtual function, i.e. between late binding and overloading, appears only when the function is invoked on an instance of D via a pointer of type C*. In [Sakkinen 92] I proposed that subclasses should not be allowed to redefine accessible non-virtual instance functions.

A second difficulty is that C++ only supports *single dispatching*, i.e. that late binding takes only the type of the "owner" object of an instance function into account. Since overloading does consider the types of all arguments, I suspect that programmers not so seldom make subtle mistakes on this point. Certainly the best-known language that supports *multiple* dispatching or "multi-methods" is CLOS [Keene 89].

As an example of the desirability of multiple dispatching, mixed arithmetic with several classes of numbers is often presented. The built-in numeric types are *not* classes in C++, but suppose that we have defined some classes like this:

```
class Number { ... }; // probably an abstract class
class Large_integer : public Number { ... };
class Rational : public Number { ... };
class Decimal_float : public Number { ... };
```

Evidently, it would be most straightforward to define the ordinary binary operators if multiple dispatching were available. Accounting for the class of the second operand by typecase programming would be a less elegant solution. Since even this is not possible in C++ (§3.6), one must, e.g., write extraneous virtual functions by which the first operand can inquire the type of the second one, in order to select the correct alternative for the actual operation.

I cannot blame C++ strongly for not supporting multiple dispatching, because most other OOPLs have not got it either, and because it is not simple to implement. Ambiguities should be treated in the same way as in static overloading to get be a consistent extension to C++. That raises a problem that does not exist in single dispatching: ambiguity detection at run time, which should probably cause an exception. To assure at compile time that this fault cannot happen would in many causes require a lot of extraneous functions to be defined — even for argument type combinations that would never occur in practice.

Two recently published approaches of multiple dispatching look well compatible with the static overloading principles of C++, although both are designed into languages very different from C++. Kea [Mugridge &c 91] is better than CLOS because it respects encapsulation and allows static checking. Although the language is applicative (functional) instead of really object oriented, its multiple-dispatch principles would appear suitable for true object-oriented languages as well.

Cecil [Chambers 92] in turn is a classless (prototype-based)[12] and highly dynamic object-oriented language. The greatest difference in multiple dispatching is the following: In Kea, even multiply-dispatched functions are ''within the encapsulation'' of only their first argument (owner), in the conventional way. In Cecil, a method has access to the private features of all the arguments on which it is dispatched, thus can ''belong'' to several objects. In C++ one could use friend declarations to achieve that.

4.2. Overloadable operators

C++ allows almost all operators to be overloaded for cases in which at least one operand is of a class type (even unions are considered classes). The number of operands, precedence, and grouping (left or right) cannot be changed, reasonably enough. Neither can totally new operators be defined: the rationale for this in [Ellis &c 90 p. 331] is sound except for the alleged syntax clashes, which look like a bogus problem to me.

To overload some operator *X* for some operand type(s) one must define a *function* with the name 'operator *X*' and with one or two arguments[13]. The operator notation is then only optional syntactic sugar: function call syntax with this peculiar function name can be used just as well. Here we have another case of two different syntaxes to accomplish exactly the same thing.

[12] A person who likes to think in terms of classes can see classes in rather thin disguise even in Cecil.

[13] If it is defined as an instance function of some class, the first operand will be an implicit argument to which 'this' points. The conditional expression operator, which is the only ternary one in the language, cannot be overloaded.

As operator overloading is only a notational convenience, the language should try to prevent its misleading use. It is indeed said in [Ellis &c 90 p. 330]:

> [...] the meaning of operators applied to nonclass types cannot be redefined. The intent is to make C++ extensible, but not mutable.

Hence the main reason why operators cannot be overloaded for *enumerations*, although that would sometimes be desirable and fully sensible, must be that enumerations are regarded as integral types (§2.2).

Unfortunately, new possibilities for truly misleading overloadings have been added in C++ Release 2.0. There are at least two standard operators in C and C++ that are fully polymorphic, i.e. applicable to operands of all types with the same semantics: the unary address-of operator '&' and the binary sequencing operator ','. I consider it a bad mistake that even these can now be overloaded. Similarly, it is unfortunate that the indirect member access operator '->' can be overloaded independently; it would be better if its meaning were always derived from the indirection operator '*' (which can also be overloaded).

It depends totally on the programmer whether any customary semantic relationships between different operators on the same type hold when they are overloaded. Suggestions to preserve some such relationships automatically were presented already in [Sakkinen 88a]. For instance, in Ada it is possible to overload the '=' operator (equal to), but the meaning of the '/=' operator (inequal to) is always automatically derived from that.

It was noted as a defect in [Sakkinen 88a] that while the prefix and postfix applications of the increment and decrement operators have different semantics for built-in types, there was no way to distinguish between them for user-defined types. This has been corrected in Release 2.1, but in an ugly way: the postfix operators must be defined as binary. I still regard my own suggestion in [Sakkinen 88a] as superior, especially since it would maintain the conventional semantic relationship between the prefix and postfix uses of the same operator.

Being only syntactically sugared functions, overloaded operators are less powerful and versatile than built-in ones on two aspects. The first aspect is that all operands of an overloaded operator are evaluated before the operator function is invoked. Therefore, while the built-in logical operators '&&' (and) and '||' (or) guarantee that their second operand is evaluated after the first one and only if necessary, no overloaded operator can do the same[14]. The same problem would affect the conditional expression operator if it were made overloadable. This is also a further objection against the overloadability of the sequencing operator.

The second aspect is that differences in the ''modes'' of operands and results — modifiable object, non-modifiable object, value — cannot be automatically taken care of by the compiler. This can be illustrated by subscripting, which is regarded as an overloadable binary operator. If T is an ordinary array, then T[i] is an object (l-value), which is modifiable if and only in T is modifiable. If the subscripting operator (bracket pair) is defined for a class, two separate operator functions are needed for the same effect. The latter one is a constant instance function (§5.3).

```
class Element { ... };
class Smart_array { ...
      Element& operator[] (int);
      const Element& operator[] (int) const;
      ... };
```

There would be means to rectify the first aspect, i.e. to pass unevaluated operands to some operators. The advantage would hardly be worth the extra complication, however. For the sake of consistent semantics, it would be an easier solution to forbid the overloading of '&&' and '||'. The second aspect is more interesting, and will be discussed in the following subsection.

4.3. Operators and references

Both the first (passing arguments) and second (returning the result) use of references in §2.5 are important for overloaded operators. If references were not regarded as datatypes, it could even be sensible to allow reference arguments and results *only* for operators, not for ordinary functions. When an overloaded operator is defined as an instance function, the first operand is automatically passed by address ('this').

The majority of the built-in operators of C++ are purely applicative: they take one or two values as operands, return one value as the result, and have no side effects. Their overloadings should preferably act similarly. If an operand of a class type is passed simply by value, side effects on the actual argument are prevented but a new object is always created for the formal argument and initialised with a copy constructor. This can be avoided, except in special situations, if the argument is

[14] This is why the equivalent short-circuit forms '**and then**' and '**or else**' in Ada are technically not classified as operators.

specified as a reference (to constant). Extraneous temporary objects and copying cannot be so easily avoided in returning the result [Ellis &c 90 §12.1.1c]. Return by reference could be a good way if C++ had garbage collection.

Some standard operators: the modifying assignment operators and the increment and decrement operators, require their first (or only) operand to be a modifiable object (non-constant l-value). Most of them return the same object (a reference to it) after modification; the postfix increment and decrement operators return a value. To achieve the same effect with the same expression syntax, the overloaded operators need to pass both the first argument and the result by reference.

A few operators return a l-value, which is modifiable or not depending on the type of the first operand. These are subscripting (§4.2) and the dereferencing operators. If subscripting is inverted (§2.4), the second operand determines the modifiability of the result. Overloaded versions should pass both the argument and the result by reference, and the modifiability should be propagated automatically.

The ordinary assignment operator is a very special case for overloading. The rules of C++ allow it to be defined only as an instance function, but it is never inherited by subclasses [Ellis &c 90 §13.4.3]. In the most common case, the result and the second operand are of the same class as the first operand, like

```
Gadget& Gadget:: operator= (const Gadget&)
```

It would not even be possible to pass either of them by value, because that would imply a recursive invocation of the assignment operator itself!

The two special restrictions on the assignment operator, at least that it should be an instance function, are quite unnecessary when the second operand is of a different type than the first one. Assigning to a class object a value of another type can have confusingly many different meanings, which we will expound in §5.1.

5. Some further subtleties

5.1. Assignment and copying

Perhaps the most complex interplay between different implicit conversions and other C++ features happens when an instance X of class A is assigned to an instance Y of another class B. The alternatives seem to be the following: (1) The appropriate overloaded assignment operator (from A to B) is invoked. (2) If class A has a conversion operator to B, it is invoked first and ordinary B assignment second. (3) If class B has a constructor with a single argument of type A, it is invoked first and ordinary B assignment second. (4) If A is a subclass of B, only the B part of X is assigned to Y (type loss of the second kind: §3.5). — Furthermore, in most cases it is important to distinguish between assignment and initialisation.

Default copy constructors and assignment operators have always been automatically generated when needed for a class, if they have not been explicitly defined by the class designer. Their working principle has been changed from bitwise to memberwise (component-wise) copying in Release 2.0. This means that if the class has any data component that itself belongs to a class with a copy constructor or assignment operator, respectively, this constructor or operator will be invoked.

The above principle is an obvious improvement: the default can be semantically adequate for many more non-trivial classes than previously. However, it sharpens the paradox with array copying mentioned in §2.3. Suppose that an array appears as a class component, e.g.

```
class Trifle { ... };
class Combo {
    Trifle misc[100];
    ... }
```

The default assignment operator of Combo *must* automatically copy 'misc' element by element (i.e. not by simple bitwise copy), at least if the elements' class Trifle has an assignment operator itself. The compiler must therefore implement such a whole-array copying operation, but it is not available to programmers. For instance, if an assignment operator must be written for Combo because the default operator is not sufficient, the programmer must code explicit loops to handle 'misc'.

If we compare C++ assignments to languages with reference semantics, we should note that those languages typically offer no operations similar to *object* assignment. A "copy" operation yields a *new* object, and is therefore the equivalent of a copy constructor in C++. Further, the automatically available operations are "shallow copy" and "deep copy" [Goldberg &c 83; Khoshafian &c 86]. Unless the class is very simple, neither of these will probably be sensible: the former is too shallow, the latter too deep.

The default functions in C++ have a much better chance of being meaningful, so that the class designer need not always write his/her own. However, there is a conceptual problem that tends to be overlooked. When we come to large application objects, copying them in *any* way may no more make sense. Take a model of the computer industry as an example: making a copy of a whole company simply has no counterpart in the real world. Neither can we sensibly make a copy of a ''person'' object in any system that handles persons as individuals. It would be nice if the class designer could specify that copying is not applicable to a certain class. The closest thing that can be done in C++ is to declare the copy constructor and assignment operator as private.

5.2. Constructors and destructors

The importance of *constructors* and *destructors* as they exist in C++ was duly recognised in [Sakkinen 88a]. I have noted afterwards, with some surprise, that equivalent facilities are missing from most other object-oriented languages. The *class body* and class parameters in Simula together are equivalent to one constructor for each class. Usually there is nothing even remotely similar to destructors, and initialisation cannot be defined as strictly as with constructors. For example in Smalltalk-80, if the **new** method of a class needs to do some non-default initialisation of the created instance, the only possibility is to define a suitable instance method; nothing protects that method later from being invoked again.

Note that the possibility of explicit deletion of objects (§3.7) is distinct from the destructor facility. A destructor guarantees that whatever *finalisation* the class designer has deemed necessary will be performed before the object is reclaimed, no matter what is the cause of the deletion. However, the execution of a destructor makes even the garbage collection of an object into a semantic event: the object does not live for ever even conceptually.

Release 2.0 has added two facilities that may be very desirable for some sophisticated special purposes, but are unwelcome for general programming. The probably more dangerous facility is that destructors can be explicitly invoked: any object can thus inadvertently be finalised many times (but ''new fashion'' destructors don't release the storage space). The other facility makes it possible to initialise an object (invoke a constructor on it) many times as well; but it requires a **new** operator to be suitably defined for the class. It is therefore less prone to be used by pure mistake.

In Simula, the possibilities of a class body are actually not restricted to object initialisation: the language supports a quasi-parallel execution of object bodies as coroutines. Most later OOPLs, including C++, do not offer this facility. However, there is a library of classes to support coroutine-style programming, which is traditionally distributed with the AT&T C++ translator [AT&T 85].

5.3. Constant objects

One anomaly noted in [Sakkinen 88a §11] was that declaring a class instance constant did not protect it against being modified by its own instance functions. The problem has now been corrected as follows. Instance functions can be declared **const**; the compiler then declares the self-reference 'this' to be of pointer-to-constant type in those functions. Only **const** instance functions can be invoked on **const** objects: see the example at the end of §4.2. — This looks like the most cautious possible policy on first sight; it ensures that constant objects can even be placed in read-only memory (if they can be initialised there).

In some other approaches, the definition of an operation as non-modifying does not absolutely prevent modifications of the object; it is the programmer's responsibility that any side effects are "benign", i.e. do not change the visible semantics of the object [Meyer 1988 §7.7]. This can actually be achieved in C++ too, because a pointer to constant can always be cast to an ordinary pointer [Stroustrup 91 p. 149]. The security is thus not absolute.

5.4. No instance-level protection

A difference between C++ and Smalltalk that often remains unmentioned even in comparative surveys is that the unit of protection is a class in C++ (and Simula), but an object in Smalltalk. Both approaches have their good and bad points: see the rationale in [Ellis &c 90 §11.2c]. At the cost of additional language complexity, it would even be possible to have both. Eiffel does that to a certain extent: only those features of a class that are exported to the class itself can be referred to between different instances of the class; direct assignment to instance variables of other objects is impossible even then. Most other statically typed languages are content with purely class-level protection.

This property of C++ is not bad in itself; but combined with the many devious ways in which one can get the address of an object, it can be surprising. The article [Liu 91] presents an interesting example on this, although I disagree with many of the paper's conclusions. It must be admitted that the instance function in the example is rather pathologic.

The idea of Liu's example is that within an array of class instances, any element may *fully legally* access any other element (including its private part); no restrictions against this can be declared in C++. We see that pointer arithmetic is in a way more harmful still in C++ than in C: the automatically defined pointer 'this' is the base from which an object can attack its neighbours. Meyer's strong opinion quoted in §1.2 hereby gets more backing.

Of course, *illegal* uses of address arithmetic can cause much worse effects, such as haphazardly overwriting data or even executable code. As argued in §2.4, it looks improbable that many C++ implementations would try to check for illegal uses of pointers.

5.5. Pointers to components (members)

A problem with function pointers remained a little open in earlier versions of C++. It is said in [Stroustrup 86 p. 153]:

> Taking the address of a member function is often useful [...] However, there is currently a defect in the language: it is not possible to express the type of the pointer obtained from this operation.

The cause was that member functions have a hidden argument (the constant pointer 'this') in addition to any explicit arguments. An implementation-dependent trick around the problem was suggested.

What I would have expected as the solution was a means to denote the types of function components, which would have been a simple matter. Instead, something much more elaborate has been introduced: there are 'pointers to members' and a couple of new operators to handle them. The chosen term is actually misleading: those are rather *component selectors* than pointers.

A "pointer to member" can be defined for both data and function components. It can be used to select between several components of the same type. It offers the same kind of dynamic component selection for objects (records) as indexing does for arrays. In the case of instance functions, there is a subtlety when a *virtual* function is selected: the pointer does not point to a fixed function, but late binding is applied when the function is invoked through it.

This concept looks suspiciously like overkill to me: it significantly increases the complexity of C++ and has very limited utility. One surprising use of pointers to virtual function components has been pointed out in [Sakkinen 92 §4.1], however. — Now we have absolutely no means to get the address of an instance function. Therefore, it is impossible to inquire e.g. whether two objects (known to have a common ancestor class) have the same binding for a given virtual function.

A pointer-to-component variable cannot be made to point to a *common* component [Ellis &c 90 §8.2.3]. Although not unreasonable, this is a little surprising, considering e.g. that common components are not separated in class declarations (§3.2). Of course, ordinary pointers can be used to refer to common components, but they can also refer to instance *data* components. The original problem affected instance functions only.

There are some further unorthogonalities yet. Although an ordinary pointer of any non-function type T* can point to an array element of type T, a component pointer of type 'T C::*' cannot point to an element of any array component of class C. Pointer arithmetic does not apply to component pointers, either (consistently with the previous restriction). Finally, there are no *references* to components.

6. Conclusion

Earlier versions of this paper have been criticised of being exhausting to read, and of lacking a clear focus and an estimation of the relative importances of the numerous observations and opinions. I am afraid that these problems have largely persisted in the revisions, and thus many readers may have skipped smaller or larger parts of the preceding text. I try to sum up the essence very compactly here.

Welsh, Sneeringer, and Hoare ended their criticism of Pascal with the following noble sentence [Welsh &c 77 p. 695]:

> It is grossly unfair to judge an engineering project by standards which have been attainable only by the success of the project itself, but in the interest of progress, such criticism must be made.

I feel that even sharp attacks on C need no such disclaimer. That language does not contain enough significant new inventions or novel combinations of old features to serve as an excuse for bad design. However, there is another excuse: C was originally designed with modest goals and for restricted purposes; the recent trend to make it into a universal programming language is harmful both to the computing world and to C itself.

A similar unfortunate development has been happening with C++. On the other hand, C++ has essentially greater merits of novelty than C in my opinion. As pointed out earlier (§3.7), C++ is an exceptional object-oriented language in allowing value semantics and direct containment for objects; some would say that it is highly *datatype complete*. Another distinguishing feature, valuable although causing a lot of complexity, is private inheritance (§3.3). A third contribution are constructors and destructors (§5.2), which protect a common heel of Achilles in OOPLs.

In the original version of [Sakkinen 88a], I bluntly punned on the name of language:

> Incrementing C by 1 is not enough to make a good object-oriented language.

That really offended Bjarne Stroustrup, and so I softened it a little in the final version. As is evident from §2, offence or no offence, today I firmly believe in the original statement: I would sharpen it further by omitting 'by 1' (although that spoils the pun). Since C supports both structured programming and strong typing only half-way, we might say that the kind of half-way object orientation that C++ offers suits the style well.

In the kind of systems programming tasks in which the low-level capabilities afforded by C++ are essential, C++ is almost as great a leap forward from C as C itself was in comparison to good macro assemblers. For this purpose, the possibility for more fine-grained programmer control over such things as structure alignment would be desirable, however; it is missing from ANSI standard C as well. C++ and other C derivatives are not necessarily the best choices available today even for systems programming, and certainly not the only choices (§1.2).

Another niche for which C++ looks very suitable is as an intermediate language: even here it is an improvement over C, which is now most commonly used. Higher-level frameworks such as Demeter [Lieberherr &c 90] can impose a lot of discipline that is missing from the programming language proper. In such environments programmers would probably not write very much ''raw'' C++ code, and the disadvantages of the language are less important than in traditional programming.

To a large proportion of people interested in C, C++, and also object-oriented programming, 'Ada' seems to be a strong curse, perhaps second only to 'PL/1' (§2.7) or 'COBOL'. One common reason for this adversity is that ''Ada is too large and complicated''. However, C++ is complex as well; as noted in [Edelson &c 89]:

> C is an elegant language; it is small and simple. The syntax of C++ is similar to that of C, but its semantics are neither small nor simple.

Everybody does not agree that even current ANSI C is small and simple. We must keep in mind that the complexity of Ada comes about largely because it tackles several important and difficult problems that C++ ignores, concurrency foremost[15]. On the other hand, Ada lacks such object-oriented features as inheritance and late binding, which are central to C++. These seem to be among the highest-priority additions suggested for the upcoming first official revision of Ada.

[15] It remains to be seen how well the new genericity features (templates) and exception handling work in practice with the rest of C++. Ada has had both facilities from the beginning.

The complexity of C++ may well have caused some erroneous interpretations of its details in this article; and some other details may already have changed in the newest versions. However, such details have little effect on the overall picture. A more serious risk is that by presenting a lot of concrete details I have made it hard for readers to see *any* overall picture.

For ordinary applications programming, more consistent and regular object-oriented languages like Eiffel look clearly preferable to C++. Not that current Eiffel can really be characterised as a *simple* language, either. I will be deeply disappointed if the bandwagon effect (§1.1) indeed makes C++ *the* object-oriented language of the 1990's. During the time of work on this article, the danger has grown quickly, and it is one reason for the harsh language that I have used.

All the above notwithstanding, C++ incorporates many fine ideas and features. Generally, those aspects of the language that are the least constrained by backward compatibility with C look the best designed. No other existing object-oriented language is perfect, either. It is to be hoped that the designers of the next generation of languages can adopt the best properties of C++ into their creations and avoid the worst ones. I am hoping above all that there *will* be a new generation of influential object-oriented languages within a few years. None of the current ones is near perfection.

Acknowledgements

Part of this work has been performed under the research project 1061120 (Object-oriented languages and techniques) of the Academy of Finland, led by Kai Koskimies (University of Tampere). The paper will be printed also as a part of my forthcoming doctor's thesis [Sakkinen 92c].

Bjarne Stroustrup (AT&T Bell Laboratories) has been helpful and communicative during the last few years, among other things by sending me copies of some references that would otherwise have been omitted. That is significant courtesy toward a clear adversary of his language. On the other hand, I have been told by one person that after reading [Sakkinen 88a] he could decide to select C++ for his applications. He had been uncertain for some time, but mine was the first *critical* article on C++ that he had seen, and he judged that the disadvantages presented there were not too severe for his organisation.

More recently, correspondence with Douglas Lea (SUNY at Oswego and NY CASE Center) has given me better insight into some aspects of C++. Commenting on the manuscript of the textbook [Budd 91] of Timothy Budd (Oregon State University) has helped in sharpening my own understanding and knowledge about C++ and its relationship to other object-oriented languages. I also thank several people at Northeastern University (at least Kenneth Baclawski, Ian Holland, and Karl Lieberherr), Peter Grogono (Concordia University), Antero Taivalsaari (University of Jyväskylä), Daniel Edelson (University of California at Santa Cruz), Tom Cargill (private consultant), Ian Joyner (Australian Centre for Unisys Software), and John Skaller (Maxtal P/L) for stimulating discussions, mostly by electronic mail.

An earlier draft of this paper has been scrutinised by Juha Vihavainen (University of Helsinki), and some useful comments have been given by Peter Wegner (Brown University). At the last stage, the anonymous reviewers made very relevant remarks about both the contents and the presentation. The occasions to give guest lectures about my ideas, e.g. at Concordia and Northeastern and at the University of Tartu (Estonia) have aided their development and formulation. Participants of the Usenet newsgroup "comp.lang.c++" and of the more recent "comp.object" and "comp.std.c++" deserve a collective acknowledgement for many good postings and interesting word battles.

References

[Abrahams 88] Paul W. Abrahams. "Some Sad Remarks About String Handling in C". *ACM SIGPLAN Notices* Vol. 23 No. 10 (October 1988), 61−68.

[AT&T 85] *UNIX System V AT&T C++ Translator Release Notes*. AT&T 1985 (307-175 Issue 1).

[Bartlett 89] Joel F. Bartlett. "Mostly-Copying Garbage Collection Picks Up Generations and C++". Technical Note TN-12, DEC Western Research Laboratory, October 1989.

[Boehm &c 88] Hans-Juergen Boehm and Mark W eiser. “Garbage Collection in an Uncooperative Envir onment”. *Software — Practice and Experience* Vol. 18 No. 9 (September 1988), 807−820.

[Budd 91] Timothy A. Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley 1991.

[Cargill 91] Tom Cargill. “Controversy: The Case Against Multiple Inheritance in C++’’. *Computing Systems* Vol. 4 No. 2 (Spring 1991), 69−82.

[Cargill 92] Tom Cargill. Private communication, 1992.

[Chambers 92] Craig Chambers. “Object-Oriented Multi-Methods in Cecil”. *ECOOP ’92 Proceedings* (Ole Lehrmann Madsen, Ed.). Springer-Verlag 1992 (LNCS 615), 33-56.

[Conradi &c 74] Reidar Conradi, Per Holager. *MARY Textbook*. RUNIT (Trondheim, Norway) 1974.

[Cox 86] Brad J. Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley 1986.

[Dahl &c 68] Ole-Johan Dahl, Bjørn Myhr haug, Kristen Nygaard. *SIMULA 67 Common Base Language*. Norwegian Computing Center 1968 (No. S-2).

[Edelson 90] Daniel Edelson. “Dynamic Storage Reclamation in C++’ ’. Technical Report UCSC-CRL-90-19, University of California at Santa Cruz, June 1990.

[Edelson &c 89] Daniel Edelson, Ira Pohl. “C++: Solving C’s Shortcomings?”. *Computer Languages* Vol. 14 No. 3 (September 1989), 137−152.

[Ellis &c 90] Margaret A. Ellis, Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley 1990.

[Goldberg &c 83] Adele Goldberg, David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley 1983.

[Grogono 90] Peter Grogono. “Issues in the Design of an Object Oriented Programming Language”. *Structured Programming* Vol. 12 No. 1 (1991), 1−15.

[Hansen 90] Tony L. Hansen. *The C++ Answer Book*. Addison-Wesley 1990.

[Johnson &c 88] Ralph E. Johnson, Brian Foote. “Designing Reusable Classes”. *Journal of Object-Oriented Programming* Vol. 1 No. 2 (June/July 1988), 22−30, 35.

[Joyner 92] Ian Joyner. “A Critique of C++’ ’. Electronically distributed report (from ian@syacus.acus.oz.au). Australian Centre for Unisys Software 1992.

[Keene 89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley 1989.

[Khoshafian &c 86] Setrag N. Khoshafian, Geor ge P. Copeland. “Object Identity”. *OOPSLA ’86 Proceedings* (Norman Meyrowitz, Ed.), *ACM SIGPLAN Notices* Vol. 21 No. 11 (November 1986), 406−416.

[Kim &c 89] Won Kim, Elisa Bertino, Jorge F. Garza. "Composite Objects Revisited". *ACM SIGMOD '89 Proceedings* (James Clifford, Bruce Lindsay, David Maier, Eds.), *ACM SIGMOD Record* Vol. 18 No. 2 (June 1989), 337−347.

[Koenig &c 90] Andrew Koenig, Bjarne Stroustrup. "Exception Handling for C++". *Proceedings of the 1990 Usenix C++ Conference*, San Francisco.

[Kristensen &c 87] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, K. Nygaard. "The BETA Programming Language". *Research Directions in Object-Oriented Programming* (B. Shriver and P. Wegner, Eds.), 7−48. MIT Press 1987.

[Lea 90] Douglas Lea. "Customization in C++". *Proceedings of the 1990 Usenix C++ Conference*, San Francisco, 301−314.

[Lieberherr &c 88] Karl J. Lieberherr, Ian Holland, Arthur M. Riel. "Object-Oriented Programming: an Objective Sense of Style". *OOPSLA '88 Proceedings* (Norman Meyrowitz, Ed.), *ACM SIGPLAN Notices* Vol. 23 No. 11 (November 1988), 323−334.

[Lieberherr &c 91] Karl J. Lieberherr, Paul Bergstein, Ignacio Silva-Lepe. "From objects to classes: algorithms for optimal object-oriented design". *Software Engineering Journal* Vol. 6 No. 4 (July 1991), 205−228.

[Liskov &c 81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, Alan Snyder. *CLU Reference Manual*. Springer-Verlag 1981 (LNCS 114).

[Liu 91] Chung-Shyan Liu. "On The Object-Orientedness of C++". *ACM SIGPLAN Notices* Vol. 26 No. 3 (March 1991), 63−69.

[MacLennan 82] B.J. MacLennan. "Values and objects in programming languages". *ACM SIGPLAN Notices* Vol. 17 No. 12 (December 1982), 70−79.

[Meek 90a] Brian Meek. "The Static Semantics File". *ACM SIGPLAN Notices* Vol. 25 No. 4 (April 1990), 33−42.

[Meek 90b] Brian Meek. "Two-valued Datatypes". *ACM SIGPLAN Notices* Vol. 25 No. 8 (August 1990), 72−74.

[Meyer 88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall 1988.

[Meyer 89] Bertrand Meyer. "From Structured Programming to Object-Oriented Design". *Structured Programming* Vol. 10 No. 1 (1989), 19−39.

[Meyer 92] Bertrand Meyer. *Eiffel: the language*. Prentice Hall 1992.

[Mugridge &c 91] Warwick B. Mugridge, John Hamer, John G. Hosking. "Multi-Methods in a Statically-Typed Programming Language". *ECOOP'91 Proceedings* (Pierre America, Ed.). Springer-Verlag 1991 (LNCS 512), 307−324.

[Nelson &c 91] Greg Nelson (Ed.). *Systems Programming in Modula-3.* Prentice Hall 1991.

[Pohl &c 88] Ira Pohl, Daniel Edelson. "A to Z: C Language Shortcomings". *Computer Languages* Vol. 13 No. 2 (July 1988), 51−64.

[Sakkinen 88a] Markku Sakkinen. "On the darker side of C++′ ′. *ECOOP'88 Proceedings* (S. Gjessing and K. Nygaar d, Eds.). Springer-Verlag 1988 (LNCS 322), 162−176.

[Sakkinen 88b] Markku Sakkinen. "Comments on "the Law of Demeter" and C++". *ACM SIGPLAN Notices* Vol. 23 No. 12 (December 1988), 38−44.

[Sakkinen 89a] Markku Sakkinen. Letter to the Editor. *ACM SIGPLAN Notices* Vol. 24 No. 3 (March 1989), 15.

[Sakkinen 89b] Markku Sakkinen. "Disciplined inheritance". *ECOOP'89 Proceedings* (Stephen Cook, Ed.). Cambridge University Press 1989, 39−56.

[Sakkinen 90] Markku Sakkinen. "On embedding Boolean as a subtype of Integer". *ACM SIGPLAN Notices* Vol. 25 No. 7 (July 1990), 95−96.

[Sakkinen 91a] Markku Sakkinen. "A paper comparison of Eif fel and C++ (and Modula-3) exceptions′ ′. workshop paper at ECOOP'91, Geneva 1991.

[Sakkinen 91b] Markku Sakkinen. "Another defence of enumerated types". *ACM SIGPLAN Notices* Vol. 26 No. 8 (August 1991), 37−41.

[Sakkinen 92a] Markku Sakkinen. "A Critique of the Inheritance Principles of C++". *Computing Systems* Vol. 5 No. 1 (Winter 1992), 69−110.

[Sakkinen 92b] Markku Sakkinen. "Corrigenda to "A Critique of the Inheritance Principles of C++"". *Computing Systems*, to appear.

[Sakkinen 92c] Markku Sakkinen. *Inheritance and other main principles of C++ and other object-oriented languages.* Dissertation manuscript, University of Jyväskylä 1992.

[Skaller 92] John Skaller. Private communication, 1992.

[Snyder 86] Alan Snyder. "Encapsulation and Inheritance in Object-Oriented Programming Languages". *OOPSLA '86 Proceedings* (Norman Meyrowitz, Ed.), *ACM SIGPLAN Notices* Vol. 21 No. 1 1 (November 1986), 38-45.

[Snyder 91] Alan Snyder. "Modeling the C++ Object Model: An Application of an Abstract Object Model′ ′. *ECOOP'91 Proceedings* (Pierre America, Ed.). Springer-Verlag 1991 (LNCS 512), 1−20.

[Stroustrup 86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley 1986.

[Stroustrup 87a] Bjarne Stroustrup. "The Evolution of C++ : 1985 to 1987". *Proceedings of the USENIX C++ Workshop.* Santa Fe, New Mexico, U.S.A. (November 1987).

[Stroustrup 87b] Bjarne Stroustrup. "Possible Directions for C++". *Proceedings of the USENIX C++ Workshop.* Santa Fe, New Mexico, U.S.A.

(November 1987).

[Stroustrup 89a] Bjarne Stroustrup. "Parameterized Types for C++' '. *Computing Systems*, Vol. 2 No. 1 (Winter 1989), 55−85.

[Stroustrup 89b] Bjarne Stroustrup. "Multiple Inheritance for C++' '. *Computing Systems* Vol. 2 No. 4 (Fall 1989), 367−395.

[Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language, Second Edition*. Addison-Wesley 1991.

[Vihavainen 87] Juha Vihavainen. *The Programming Language Mode : Language Definition and User Guide*. University of Helsinki 1987.

[Wachowitz 91] Marc Wachowitz. Private communication, 1991.

[Waldo 91] Jim Waldo. "Controversy: The Case For Multiple Inheritance in C++". *Computing Systems* Vol. 4 No. 2 (Spring 1991), 157−171.

[Welsh &c 77] J. Welsh, W.J. Sneeringer, C.A.R. Hoare. "Ambiguities and Insecurities in Pascal". *Software - Practice and Experience* Vol. 7 No. 6 (November / December 1977), 685−696.

[Wirth 88] Niklaus Wirth. "The Programming Language Oberon". *Software - Practice and Experience* Vol. 18 No. 7 (July 1988), 671−690.

YHTEENVETO (FINNISH SUMMARY)

Tämä tutkimus on artikkeliväitöskirja, jonka luvut 2 □ 6 ovat eri yhteyksissä ilmestyneitä tai ilmestyviä konferenssi- ja aikakauslehtiartikkeleita (2 ja 3 vuonna 1988, 4 vuonna 1989, 5 ja 6 vuonna 1992). Pidin lukijoiden kannalta tärkeänä saada ne mukaan itse väitöskirjaan, ja kiitän alkuperäisiä kustantajia uudelleenjulkaisuluvista, samoin Jyväskylän yliopiston julkaisutoimikuntaa ja sarjan toimittajaa Airi Salmista suostumisesta tähän ratkaisuun. Luku 1 sisältää sekä johdannon tutkimusaiheeseen, muita lukuja koskevia jälkihuomautuksia (kohta 2) että viitteitä tutkimuksen mahdollisiin lupaaviin jatkoaiheisiin.

Tutkimusalueen suomenkielinen sanasto on vielä vakiintumatonta (monia englantilaisiakin termejä käytetään kirjallisuudessa hieman vaihtelevissa merkityksissä). Ainoa tuntemani pätevä suomenkielinen johdanto olio-ohjelmointiin [Vihavainen 89] on varsin suppea, ja tietotekniikan arvovaltaisin sanastoteos [Atk 90] sisältää vain muutamia oliokeskeisyyden erikoistermejä. *Object* on kuitenkin saanut tässä erikoismerkityksessä täysin vakiintuneen vastineen 'olio'. Termin *object oriented* suomennoksena käytän tässä sanaa 'oliokeskeinen' (koska 'olioperustainen' [Atk 90] toisi mieleen englanninkielisen termin *object based*, jolle usein halutaan antaa oma merkitysvivahteensa). *Class* on luonnollisesti 'luokka' ja *inheritance* 'periytyminen' tai 'perintä'.

Luvun 1 kohdassa 1 esittelen ja arvioin muutamia tunnettuja oliokeskeisyyden määritelmiä (ks. lukua 4). Erityisesti perustelen, miksi en pidä 'sanomanlähetykseen' liittyvää terminologiaa onnistuneena. Esitän sitten hyvin väljän määritelmän: Oliokeskeinen ohjelmointikieli tukee tiedon ja käyttäytymisen kapselointia[1] olioiksi, joilla on vahva identiteetti ja eheys ja joissa on tiedonkätkemisen mahdollisuus. Normaalisti se sallii periytymisen ja siihen liittyvän operaatioiden dynaamisen sidonnan; periytyminen voi olla joko olioiden tai olioluokkien välistä. Normaalisti kieli sallii myös olioiden luomisen, muuntamisen ja tuhoamisen.

Luvun 1 kohdassa 3 mainitsen muutamia periytymisen mielenkiintoisia aspekteja, joita ei juuri käsitellä muissa luvuissa. Erityisesti *ankara* periytyminen (perittyjen ominaisuuksien muuttaminen ei ole sallittua) ja ns. "sekoitusperiytyminen" (mix-in inheritance) vaikuttavat lupaavilta. Totean normaalin (ei ankaran) *periytymisen perusdilemman*: jotta jonkin operaation (rutiinin, "menetelmän") toiminta olisi ymmärrettävissä ja hallittavissa sen luokan yhteydessä, jossa se määritellään, sen pitäisi pe-

[1] Kapseloinnilla en vielä sinänsä tarkoita tiedonkätkemistä tai tietoabstraktiota.

riytymisen yhteydessä muuttua mahdollisimman vähän; jotta se sopisi hyvin aliluokkiin, sen pitäisi muuttua huomattavan paljon automaattisesti. Luvussa 3 korostetaan lähes pelkästään edellistä puolta mutta monesti kirjallisuudessa (esim. William Cookin ansiokkaassa väitöskirjassa, johon luvussa 1 viitataan) yhtä vahvasti jälkimmäistä.

Luvun 1 kohdassa 4 mainitsen muutamia mielenkiintoisia aiheita, joiden suuntaan tutkimusta voisi jatkaa tästä väitöskirjasta: olioiden olemassaoloriippuvuudet, operaatioiden paikallisuuden takaaminen, luokka- ja prototyyppilähestymistavan yhteensovittaminen ym. Useimpia näistä aiheista olen jo alustavasti käsitellyt, pääasiassa julkaisemattomissa käsikirjoituksissa.

Luvussa 2 analysoidaan C++-kieltä osittain sellaisena, kuin se oli käytettävissä vuoden 1988 alussa (versio 1.2), osittain ottaen huomioon tärkeitä muutoksia ja uusia ominaisuuksia, joita oli jo silloin esitelty artikkeleissa; uusi versio 2.0 julkistettiin kuitenkin vasta kesällä 1989. Suurimmalta osalta tämän luvun sisältö on edelleenkin ajankohtaista, vaikka useita siinä mainittuja virheitä ja ongelmia on myöhemmin korjattu, osittain jopa suurin piirtein ehdottamallani tavalla.

Näkemykseni mukaan C++:n perusvika on sama kuin sen perusvahvuus: lähes täydellinen yhteensopivuus C:n kanssa. Luvussa 2 (kohdissa 2 ja 4) esitän muutamia syitä, miksi C on sopimaton korkeatasoisen oliokeskeisen kielen peruskieleksi. Keskeisin syy on liian hallitsematon osoittimien ja taulukoiden käsittely; se on niin oleellinen osa C:tä ja tyypillistä C-ohjelmointia, että yhteensopivuuteen pyrkivässä kielessä sitä ei voisi ajatella muutettavan ratkaisevasti.

Luvussa 2 mainitaan myös useita C++:n olio-ominaisuuksien puutteita. Tärkeimmät niistä liittyvät dynaamisen muistin käsittelyyn (kohdat 6 ◘ 8). Versiossa 2.0 käyttöönotetut periaatteet toisaalta muistin varaamisen ja olion rakentajan (constructor), toisaalta muistin vapauttamisen ja olion tuhoajan (destructor) välisestä yhteydestä ovat korjanneet tilannetta oleellisesti. Silti edelleenkään ei ole mahdollista määritellä luokkia, joiden ilmentymät (oliot) olisivat keskenään erikokoisia. Luvun 2 kohdassa 12 annettu ohje tällaisten tarpeiden toteuttamiseksi on siis yhä hyödyllinen.

Luvussa 2 ehdotetaan keinoja, joilla luokkaoperandeille uudelleenmääriteltyjen eri operaattorien merkitysten välillä saataisiin automaattisesti säilymään samanlaisia yhteyksiä, kuin niiden välillä vallitsee normaalisti, esim. '+', '+=', '++'. Tässä suhteessa C++:n uusimmissa versioissa on päin vastoin menty huonompaan suuntaan: jopa joitakin sellaisia operaattoreita, joilla luonnostaan olisi täysin yleinen merkitys, voidaan luokkien yhteydessä määritellä mielivaltaisesti uudestaan (luku 6).

Luvussa 3 tutkitaan Karl Lieberherrin ja muiden Northeastern-yliopiston tutkijoiden vuonna 1988 julkaisemaa olio-ohjelmoinnin tyylisääntöä, Demeterin lakia. Sen alkuperäinen muotoilu oli hyvin yleinen mutta ensi sijassa Flavors-kielen tyypitettyyn laajennokseen sopiva. Tässä pyrin ottamaan huomioon C++:n erikoisominaisuudet, mm. sen, että kaikki C++:n "oliot" eivät ole luokkaolioita (koska C++ on hybridikieli).

Tutkin Demeterin lakia kriittisesti (osittain liioitellunkin kriittisesti) myös muista näkökulmista. Laki estää kunkin luokan operaatioita näkemästä tai ainakin hyödyntämästä minkään muun luokan rakennetta ja omankin luokkansa *osien* rakennetta. Lisäksi laki rajoittaa vahvasti sitä, minkä muiden luokkien operaatioita kukin operaatio ylipäänsä saa kutsua. Jälkimmäiseen osaan ehdotin sellaista lievennystä, että luokka voitaisiin tarvittaessa julistaa toisen luokan "tuttavaksi". Tämä ehdotukseni onkin otettu huomioon Demeterin lain myöhemmissä versioissa.

Luku 3 oli alun perin ilmestyessään (lehtiartikkelina) minulle tärkeä siksi, että se aloitti edelleen jatkuvan hedelmällisen yhteyden Lieberherrin Demeter-ryhmään. Heidän avoin asenteensa kärkeväänkin rakentavaan kritiikkiin oli yhteistyön oleellinen edellytys, ja siitä olen yrittänyt ottaa oppia.

Luvussa 4 esitetään lievästi provokatiivinen luettelo olioiden ja sitä kautta oliokeskeisten ohjelmointikielten tärkeimmistä ja vähemmän tärkeistä ominaisuuksista: periytyminen luetaan vähemmän tärkeiden joukkoon. Tarkoituksena on korostaa sitä, että monia oleellisia seikkoja ei kirjallisuudessa tavallisesti mainita eksplisiittisesti, mutta silti periytymistä ei saisi määritellä eikä toteuttaa niin, että esim. olioiden identiteetti ja eheys kärsivät (ks. tiivistelmän alkupäässä olevaa määritelmää).

Tässä luvussa jaetaan periytyminen *oleelliseen* (essential) ja *satunnaiseen* (incidental) lajiin, joista jälkimmäinen vastaa pelkän toteutuksen perimistä tai koodin uudelleenkäyttöä. Oleellisen perimisen äärimmäinen tapaus on ali- ja yliluokan välinen *täydellinen yhteensopivuus*, joka oli jo aikaisemmassa kirjallisuudessa osoitettu erittäin tiukaksi, tyypillisissä luokissa harvoin toteutuvaksi vaatimukseksi.

Luvussa 4 pyritään periytyminen selittämään ja mallintamaan mahdollisimman suurelta osin koostamisena. Varsinkin satunnaisen periytymisen ja tavallisen koostamisen välinen ero osoitetaan asteittaiseksi (kohta 6). Tosin tällöin vaaditaan, että satunnaisessa periytymisessä ei sidota virtuaalisia operaatioita dynaamisesti. Dynaaminen sidonta tapahtuu kuitenkin esim. C++:n yksityisessä periytymisessä. Nykyisin katsoisinkin sen mahdollisuuden periytymisen ja pelkän koostamisen erottavaksi tunnusmerkiksi. — Myös toisen ääripään, täydellisesti yhteensopivan periytymisen, selittäminen koostamisen avulla onnistuu hyvin (kohta 7). Välialue jätetään suurelta osin myöhemmän tutkimuksen kohteeksi.

Luvun 4 kohdassa 8 tutkitaan erityisesti *haarautuvaa* ("fork-join") moniperintää. Siinä osoitetaan, että oliokielissä tyypillisesti noudatetut moniperinnän periaatteet (esimerkkinä Eiffel-kieli) eivät tällaisessa tapauksessa takaa yliluokka-osaolioiden eheyttä, jota pidän hyvin tärkeänä asiana. Sen sijaan C++:n moniperintä, joka tätä lukua kirjoitettaessa oli uutuus, säilyttää osaolioiden eheyden.

Luvussa 5 sitten löydän runsaasti vikoja myös C++:n periytymisperiaatteista. Ne esitetään lyhyesti kuutena teesinä, joista yhteen liittyy myös korollaari. Useimmat ongelmat voidaan johtaa kahteen perusvirheeseen. Ensimmäinen perusvirhe koskee myös yksittäisperintää: virtuaalioperaatioiden uudelleenmääriteltävyys on liian rajoittamaton. Toinen perusvirhe liittyy haarautuvaan moniperintään: yliluokkaosien *jaettavuus* ("virtuaalisuus") on liian globaali.

Toisen perusvirheen täydelliseksi korjaamiseksi esitän luvun 5 kohdassa 5.4 säännön, joka on koko luvun tärkein anti. Tämän säännön perusteella yliluokkaosien erityinen määritteleminen jaettaviksi tai *monistettaviksi* on periaatteessa tarpeetonta — jakaminen tai monistuminen määräytyy sen mukaan, mitkä periytymissuhteet ovat *julkisia* ja mitkä *yksityisiä*. Sääntöön liittyy kaksi hyväksyttävien periytymisrakenteiden rajoitusta; korjauksissa (luvun lopussa) sitten jälkimmäinen rajoitus näytetään tarpeettomaksi, mutta ensimmäinen on oleellinen.

Luku 5 on hyvin konstruktiivinen: havaittujen virheiden korjaamiseksi siinä ehdotetaan täydellisiä uusia periytymissääntöjä, jotka poikkeavat nykyisistä mahdollisimman vähän. Ne eivät luultavasti haittaisi olemassa olevia ohjelmia kovin paljon. Pidän selvänä, että monimutkaisia moniperintäverkkoja ei voida käyttää mielekkäästi, jos C++:n nykyiset säännöt pysyvät voimassa. Osa ehdotuksista voidaan kuitenkin toteuttaa ohjelmointikurin avulla muuttamatta itse kieltä. — Edellä mainittujen ensimmäisen ja toisen perusvirheen yhteisvaikutuksena esitetään kohdassa 5.6 yllättävän patologinen tilanne, jota kutsun "eksponentiaaliseksi jojoilmiöksi".

Yksi C++:ssa esiintyvien ongelmien aiheuttaja on se, että kielen suunnittelussa ja kehityksessä toteutusnäkökohtia on painotettu liikaa semantiikan ja käsitteellisen mallintamisen kustannuksella. Luvun 5 käsittelemissä periytymisongelmissa tämä näkyy siinä, ettei *yksityisen* periytymisen merkitystä ole mietitty loppuun asti. Useimpia ongelmia ei ollenkaan syntyisi, jos kielessä olisi vain julkinen periytyminen. Sinänsä yksityinen periytyminen kuuluu C++:n huomionarvoisiin erikoisominaisuuksiin, vaikka se lisää kielen monimutkaisuutta. Vastaavaa mahdollisuutta ei juuri missään muussa oliokielessä ole.

Luku 6 pyrkii tutkimaan C++:n nykyisiä epäkohtia hyvin laajalti; pelkästään periytymiseen liittyvät seikat on erotettu edelliseen lukuun, joka on huomattavasti yksityiskohtaisempi ja enemmän C++:n tuntemusta edellyttävä kuin tämä. Luku 6 on selvästi asenteellinen, vaikka siinä mainitaan useita C++:n hyviäkin ominaisuuksia. Syynä on se, että C++ on viime vuosina ollut nousemassa olio-ohjelmoinnin kansainväliseksi standardikieleksi; siinä on kuitenkin niin paljon vikoja, että tällainen kehitys on erittäin haitallista.

Tärkein tässä luvussa korostettu asia, joka oli vielä luvussa 2 jäänyt vähälle huomiolle, on C++:n *oliokeskeisyyden perusvika* (kohta 3.5): oliot eivät sisällä riittävää ajonaikaista tyyppitietoa, päin vastoin kuin käytännöllisesti katsoen kaikissa muissa oliokielissä. Tämä puute on erityisen vaarallinen yhdessä osoittimien hallitsemattomuuden kanssa, jota käsitellään tässä luvussa vielä lisää (vrt. lukuun 2). Toteankin sarkastisesti (kohdassa 6), että koska C++ on C:n perintönä sekä heikosti tyypitetty että heikosti rakenteinen (kohta 2.7), niin heikko oliokeskeisyys sopii hyvin tyyliin.

Ajonaikaisen tyyppitiedon puuttumisesta seuraa mm., että olioiden tyyppi ei voi vaikuttaa ohjelman valintoihin muuten kuin virtuaalioperaatioiden myöhäisen sidonnan kautta. Ns. "typecase"-ohjelmointi on siten mahdotonta; vaikka sen liikakäyttö on houkuttelevaa ja haitallista, joissakin tilanteissa se olisi erittäin tarpeellista (kohta 3.6). — Läheisesti tyyppitiedon puuttumisen liittyy myös se, että kompleksiolioiden identiteetti (esim. moniperiytymisen yhteydessä) on heikko (kohta 3.8).

Tässä luvussa käsitellään myös C++:n *viitetyyppejä*, jotka ovat hieman eri asia kuin osoitintyypit (kohdat 2.5, 4.3). Arvioin ne puolinaisiksi tietotyypeiksi, joilla pyritään korvaamaan viiteparametrien puuttuminen. Ne lisäävät osaltaan kielen vaikeutta ja monimutkaisuutta. C++:aa mutkistavat tuntuvasti myös monet uusissa versioissa lisätyt ominaisuudet. Jotkin niistä, kuten moniperintä, ovat tärkeitä ja hyödyllisiä. Jotkin taas ovat tarpeellisuudeltaan kyseenalaisia: esim. osoittimet komponentteihin (kohta 5.5).

Yksi osoitinaritmetiikan ja olio-ominaisuuksien yhteisvaikutuksena syntyvä ongelma olisi vielä ansainnut maininnan tässä luvussa: luokkatyyppiin määritelty osoitin voi osoittaa myös aliluokan oliota, mutta tällöin osoitinaritmetiikka ei olekaan laillista (luvun 1 kohta 2.6).

Luvun lopuksi (kohdassa 6) väitän, että atk-maailma tarvitsee pian uusia, selvästi nykyistä sukupolvea parempia olio-ohjelmointikieliä. Muutkaan nykyiset kielet kuin C++ eivät näytä olevan vielä tarpeeksi hyviä kelvatakseen pitkäaikaisiksi standardivälineiksi. Ohjelmointikielten tähänastisen historiankaan perusteella en uskalla kuitenkaan olla kovin optimistinen.

Viitteet

[Atk 90] *Atk-sanakirja — Finnish Dictionary of Information Processing*, viides korjattu painos. Suomen Atk-kustannus Oy 1990.

[Vihavainen 89] Juha Vihavainen. “Olio-ohjelmointi”. Osa I: “Oliot ja luokat”. *Dimensio* 53:7 (1989), 29 □ 36. Osa II: “Oliokielten mekanismit”. *Dimensio* 53:8 (1989), 29 □ 36. Osa III: “Oliokielten vertailua”. *Dimensio* 53:9 (1989), 29 □ 36.