The darker side of C++ revisited

Markku Sakkinen
Department of Computer Science and Information Systems
University of Jyväskylä
PL 35, SF-40351 Jyväskylä, Finland
Internet: sakkinen@jytko.jyu.fi Bitnet: SAKKINEN@FINJYU

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++.

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 superclass 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 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].

[1] The term 'template' seems to have a long tradition in this meaning: it is used already in [Conradi &c 74].

[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.

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 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 overused. 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 overdriven, 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.

[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].

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 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

[4] Pure syntax issues were not treated at all in [Sakkinen 88a].

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.

[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.

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 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 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 "mainstream" 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

[6] "All programming languages (not only assembly languages!) should support macroes." [Conradi &c 74].

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, instance function
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 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 superclasses 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 from [Sakkinen 92a]. Already in early versions of C++ where 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 introduced for allowing access to protected components. After the access rules of protected components were 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 required for a class or function to access even public components of another class, unless it has access rights according to the basic Law of Demeter [Lieberherr &c 88]. (This was a relaxation 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 superclass is not duplicated. On the other hand, arbitrary conversions between pointer types are possible by explicit casts. Obviously, 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_something' 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 superclass (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 superclass (§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 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

[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.

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 programming languages, as well as LISP and CLU [Liskov &c 81], are based on *reference semantics*: variables cannot contain objects (except some atomic values) but only pointers to them. This means in practice that all objects are 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 Eiffel, one can define ‘‘expanded types’’ [Meyer 92 §12.2] for such behaviour.

A consequence of reference 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 there is no GC but programmers must take care to explicitly delete objects that are no more needed. However, this problem does not concern at all those objects that are allocated on the stack or statically; the *need* for garbage collection is not so great 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] reports on an approach that succeeded remarkably well in adding GC to existing C (not even C++) software 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 restricting 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.

[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.

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 forever; 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.

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.

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

[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.

[14] This is why the equivalent short-circuit forms '**and then**' and '**or else**' in Ada are technically not classified as operators.

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 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 forever 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.

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.

Demeter is a trademark of Northeastern University. **Eiffel** is a trademark of The Non-profit International Consortium for Eiffel (NICE). **Objective-C** is a trademark of Stepstone Corporation. **Simula** is a trademark of Simula a.s. **Smalltalk-80** is a trademark of ParcPlace Systems, Inc. **UNIX** is a trademark of AT&T.

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.

[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.

[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 Weiser. “Garbage Collection in an Uncooperative Environment”. *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 Myhrhaug, 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) 1992.

[Keene 89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley 1989.

[Khoshafian &c 86] Setrag N. Khoshafian, George 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. Nygaard, 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 Eiffel 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 JyvÌskylÌ 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. 11 (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.