On the darker side of C++

Markku Sakkinen
Department of Computer Science, University of Jyväskylä
Seminaarinkatu 15, SF-40100 Jyväskylä, Finland
Electronic mail: markku@jytko.jyu.fi

Abstract

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

1. Introduction

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

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

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

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

2. About object orientation and language extensions

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

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

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

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

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

Extending some existing language with *lower-level* capabilities is not very difficult in general. The extreme in this direction is the escape to assembler, which exists in several languages or implementations.

But when someone sets out to enrich an existing language with object-oriented or other *higher-level* features, trying to keep totally upward compatible with the base language can be problematic. Obviously, it is easier to extend a language that seems too restricted (e.g. Pascal) than one that has very general, powerful, and accordingly error-prone facilities (e.g. C). One recent example of extending Pascal in an object-based direction (in CLU fashion) is presented in [Saje, Olsz].

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

3. Miscellaneous problems inherited from C

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

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

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

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

4. Array problems

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

typedef basetype atype [dimension];

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

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

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

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

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

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

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

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

5. Classes

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

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

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

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

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

```
class myclass;
```

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

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

Within a member function, there is always an implicit parameter **this**, which is a pointer to the class instance whose component the function is invoked as. Perhaps a little surprisingly, a member function of

myclass, say, *can* be called even without an instance of *myclass*:

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

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

6. Problems with constructors and destructors

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

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

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

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

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

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

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

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

7. Mistakes with derived classes

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

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

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

Correspondingly, it says (in §8.5.7):

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

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

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

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

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

8. A problem with virtual functions

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

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

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

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

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

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

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

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

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

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

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

9. Some suggestions to cope with the problems

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

As already mentioned in § 6, it will become possible to overload the **new** and **delete** operators for a class. Very importantly, the **new** operator function will get as one parameter the size of the object to be

allocated. It is thus possible to write an allocator for a base class that will work correctly for any derived class also. Alternatively, one may write an overriding allocator for some derived class if needed. In either case, the appropriate version of **new** will be called before any constructor and so the need to assign to **this** within constructors disappears. Hence, the constructors can really be invoked in the order described in § 7.

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

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

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

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

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

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

10. Operator overloading

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

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

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

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

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

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

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

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

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

11. Constants and pointers to constants

C++ allows one to derive a constant type from any non-constant data type. This general 'constant' concept is very useful, exists in many other languages, and is unambiguously defined when applied to "pure data" types. However, when the base type is a class with member functions, there arises a problem that none of the references has observed: what member functions, if any, of a constant class instance should be callable? The present implementations appear to allow all member functions to be called,

including the assignment operator if it is overloaded. A real solution to this problem would require, either an explicit declaration of those member (and friend) functions that are allowable with constant class objects, or disproportionate run-time effort, at least on typical current computers (with a truly sophisticated hardware architecture like that of Burroughs, it could be easier). We should thus only warn programmers not to trust "constants" of any class that has any modifying member or friend function.

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

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

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

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

12. Some practical difficulties and hints

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

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

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

values wrong; current versions handle them correctly in *many* cases, but a function that returns an abnormal class value may again cause mysterious errors with some future release.

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

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

13. Conclusion

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

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

Acknowledgements

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

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

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

UNIX is a trademark of AT&T. Ada is a registered trademark of the United States Department of Defense. Smalltalk-80 is a trademark of Xerox Corporation. VAX/VMS is a trademark of Digital Equipment Corporation.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Typing is strong as a rule (cf. § 3), but there are some implicit type conversions (cf. example in § 6). Furthermore, explicit type conversions or casts can be effected very generally between types; even if *atype* cannot be

converted to *btype*, at least *atype** can be converted to *btype**. The example of § 8 uses traditional C cast notation in the constructor function *flexstring::flexstring* to convert a **char*** value to *flexstring**. Whenever the target type can be expressed as just a type name, functional notation can be used as well:

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

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

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

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

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

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

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

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

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

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