The ModulaTor

Oberon-2 and Modula-2 Technical Publication

Erlangen's First Independent Modula_2 Journal! Nr. 9, Oct-1994

Oberon vs. C++

by Josef Templ

originally published in German by Heise Verlag, iX Sept. 94, Germany; translated to English by Josef Templ.

While C++ is gaining acceptance in the software industry, Oberon is going to replace Pascal for educational purposes. A comparison of both languages shows their concepts and differences.

When Niklaus Wirth, well known for the development of Pascal and Modula-2, and Juerg Gutknecht in the mid eighties started to develop a new operating system for personal computers, the existing programming languages turned out to be insufficient for the development of extensible software systems [1]. Even the monster languages such as PL/I or Ada could not be used to construct robust and reliable programs that can be extended later on. Small languages such as Pascal, Modula-2 or C also had - despite many successful concepts - deficiencies in their type systems to stand the test of this new software engineering challenge.

Niklaus Wirth recognized this demand for 'rightsizing' and developed a n language which was supposed to have greater expressivity and at the same time should be even easier to learn and use than its predecessors. The result of this effort is the programming language Oberon [2]. A comparison with the language C++ [3], which has been developed by Bjarne Stroustrup about the same time for about the same reasons as a successor of C, should highlight the characteristics of both languages. A comparison seems to be admittable because both claim to be general purpose, strongly typed, object-oriented and efficiently implementable.

C++ is almost an industry standard now although the existing implementat deviate significantly from each other. Oberon has become established - just like Pascal 20 years ago - in the academic and educational environment first. This has been supported by freely available language implementations for all major hardware and software platforms including DOS, Windows 3.1, NT, OS-2, AIX, Solaris, Ultrix, Irix, HP-UX and Linux.

Data abstraction by means of modules

Syntactic differences are not the criterion for this comparison. We try focus on the abstraction mechanisms and the safety, a language provides. Abstraction means 'omission of details'. It is the most important means for mastering complexity. With respect to programming, it means to ignore the implementation and to consider the specification or interface only. Oberon programs consist of a set of modules which interact via an import/export- mechanism and allow to hide their implementation from clients. Modules are in Oberon the means for expressing concepts such as abstract data structures, abstract data types or simply to realize function libraries. Modules also serve as compilation units and in the Oberon system as units of system extension, i.e. a module can be loaded on demand during runtime and thereby extends the running program by new functionality. Furthermore, a user may invoke exported parameterless procedures directly as a command. A module may also contain a body, which is typically used to initialize global variables. Listing 1 shows an example. The import clause lists all imported and thu usable modules. A client of M can only use those objects which are marked for export by a '*' following the object's name (e.g. T*). To distinguish objects with the same name imported from different modules, Oberon requires to prefix imported names by the name of the exporting module (e.g. M.P). This avoids ambiguities and helps in reading programs since it is always made explicit where an object has been defined.

```
MODULE M;
IMPORT M1, M2 := MyModule;

TYPE
    T* = RECORD
            f1*: INTEGER;
            f2: ARRAY OF CHAR
    END ;

PROCEDURE P*(VAR p: T);
BEGIN
    M1.P(p.f1, p.f2)
END P;

END M.
```

Listing 1: Example of an Oberon module: The import clause lists all imported modules. The exported objects are marked with '*'.

Imported modules may be renamed in the import clause in order to abbrevi long names or to experiment with different variants of a module without too many changes in the client (e.g. M2 := MyModule). Record fields may be exported selectively (e.g. f1*), i.e. it is possible to keep some fields private while others are exported.

C++ simulates modules via the preprocessor

C++ does not have a module concept in the language proper but simulates it in the well-known way via the C-preprocessor (cpp) and appropriate programming conventions (header files). The global name space that C++ inherited from C does not preclude name clashes during the linking step. To avoid this problem, classes are sometimes used to simulate the name scope of modules. In the case of interrelated classes or procedures which refer to more than one class, so- called friends must be used, which are sort of a scope-goto - a construct that allows to circumvent the usual scoping rules of the language. Friends make names visible where they would not be visible otherwise. Since this mechanism is still unsatisfying for large software systems, extensions to the scoping mechanisms - namespaces - are beeing discussed by the C++ standardization committee [5]. However, namespaces still depend on the C-preprocessor, thus, they cannot be regarded as a proper module concept in the language.

Another often cited criticism of C++, the missing initialization order, also solved by Oberon's modules. In contrast of cpp's include mechanism, the import relationship forbids cycles. Thus, the imported modules can always be initialized before their clients.

For system-level programming, Oberon offers the pseudo module SYSTEM, wh provides implementation and machine dependent operations. Modules which import SYSTEM are inherently unportable and unsafe but easily identified by the word SYSTEM in their import list. C++ allows the usage of system level operations without specially marking such programs. When porting programs from one machine to another, this might lead to unpleasant surprises and long debugging sessions.

Safety in programming languages

Nowadays nobody expects that an electric shaver can be plugged into a high-voltage socket. Furthermore, for the case of a short circuit or similar malfunctioning of a correctly connected appliance there are additional fuses. Surprisingly, these concepts of safety are not well-established in most programming languages. Of course, not every programming error can be precluded by the design of a programming language. Nevertheless, the avoidance of certain error classes and the detection of runtime errors are important quality aspects [4, 5]. Both Oberon and C++ rely on the notion of strong typing. The approach to that, however, is quite contrary. In Oberon (as in Pascal) a variable is associated with an arbitrary complex type, in C++ (as in C) a type is associated with an arbitrary complex designator (lvalue). This lvalue acts as a prototype for the usage of the variable and defines the variable's type implicitly. By inverting the declaration and isolating the variable, the variable's type can be reconstructed. A concrete example is the definition of a pointer v to a structure x as in:

```
struct x *v;
```

This means that the lvalue *v is of type struct x. '*' denotes dereferenciation, therefore the type of v can be deduced as pointer to struct x. In Oberon one would write

```
VAR v: POINTER TO x;
```

The variable v in this declaration is already isolated. In case of more complex declarations, Oberon's approach is definitely simpler and more regular. Eventually, in both languages a type is associated with every variable, which defines the set of values and applicable operators. By that, many erroneous usages of variables and procedures can be detected before program execution and help to avoid mysterious program crashes.

Pointer arithmetic in C++ is dangerous

For those errors that cannot be detected before program execution, Oberon goes one step further by guaranteeing type safety and memory consistency even at run time. The necessary fuses, for example for array-bound checking, can be implemented with almost no overhead in execution time and program size. C++ defines an array as identical with a pointer to the first element and allows pointer arithmetic. This precludes index checking in practice. A further safety loophole in C++ exists in the management of dynamic storage where Oberon still guarantees memory consistency by means of automatic garbage collection.

In contrast to BASIC an most scripting or fourth generation languages bo Oberon and C++ offer the possibility to construct dynamic data structures which are interrelated by means of pointers. Such structures not only grow but also shrink. In the latter case, the C++ programmer has to explicitly free the unused storage. To support this task, C++ offers the notion of destructors, which are automatically activated whenever an object is deallocated.

Destructors, however, do not solve the problem that objects are dealloca too early or too late. Many hours of debugging time have already been spent to detect and fix such errors. In vain for extensible programming systems. It can easily be shown that the programmer cannot know the correct time of freeing an object in this case. Therefore, and not only for convenience, Oberon relies on a conceptually infinite heap storage, which only allows to allocate but not to deallocate objects.

Oberon with integrated garbage collection

In contrast to the programmer, the runtime system can easily decide, when an object is no longer in use and deallocate the associated storage. This technique, also called automatic garbage collection, implements the illusion of an infinite heap and leads to a significant gain in productivity. Probably, it is garbage collection and not the syntax that attracted the users of languages such as Smalltalk or Lisp. It is not surprising that introduction of garbage collection is a hot topic within the C++ community. Due to pointer arithmetic, however, it is much more difficult if not impossible to introduce it in C++.

OOP-concept is record extension

Roughly speaking, both Oberon and C++ are object-oriented extensions of existing languages. The approaches to this, however, are fairly different. C++ essentially supports object-oriented programming (OOP) a la Simula-67, Oberon does not suggest a particular OOP-style but leaves it to the programmer to select the appropriate technique for a given task. All these techniques are based on the notion of record extension, which replaces the variant records (Unions in C) of its predecessors. Record extension means that a new record type can be defined as an extension of an existing one.

The base type and the extended type are upward compatible to each other, all operations which can be applied to the base type can also be applied to the extended type but not vice versa. Two fundamental OOP-styles can be identified in Oberon. They are distinguished by the fact that a message is represented explicitly as an Oberon data structure or implictly as a procedure call.

In the first case, messages are represented as records (message records) are passed explicitly to a procedure (the message handler) as variable parameters. The handler is typically bound to the receiving object by means of a procedure variable (c.f listing 2). Objects are usually allocated on the heap and referenced via pointers.

```
TYPE
     Object = POINTER TO ObjectDesc;
     ObjectMsg = RECORD END ;
     Handler = PROCEDURE (O: Object; VAR M: ObjectMsg);
     ObjectDesc = RECORD
             handle: Handler
     END ;
```

Listing 2: Message records are explicitly passed to the handler procedure, which is bound to the receiving object by a procedure variable.

Applying record extension to messages it is possible to create a hierarc message types. The message type DisplayMsg, for example, is derived from the base type ObjectMsg (c.f. listing 3). Further specialization of DisplayMsg is possible. The handler distinguishes different message kinds by means of the type test operator IS and responds in an object-specific way to the message. Using message records and handlers seems to be rather inconvenient and inefficient at the first glance. However, they do have certain advantages as well, which explains why they are the dominant OOP-style in the Oberon system.

The advantages are:

Messages can be introduced where they are needed. It is not necessary to declare them together with the base type.
Messages can be handled generically without knowing their type or interpreting their contents. A container object, for example, can forward messages to its members without knowing all these messages. Generic broadcast, forwarding and delegation is possible.
The effect of extensible parameter lists can be achieved by extending message records.
There is a clean separation between subtyping (record extension) and subclassing (code inheritance). Code inheritance including multiple and even dynamic inheritance can be achieved by programming an appropriate message dispatching mechanism in the handler (c.f. ELSE branch in listing 3).

```
TYPE
     CopyMsg = RECORD (ObjectMsg)
             deep: BOOLEAN;
             cpy: Object
```

```
        END ;

        DisplayMsg = RECORD (ObjectMsg)
                F: Frame;
                x, y: INTEGER
        END ;

    PROCEDURE HandleMyObject (O: Object; VAR M: ObjectMsg);
    BEGIN
        IF M IS CopyMsg THEN ...
        ELSIF M IS DisplayMsg THEN ...
        ...
        ELSE Objects.Handle(O, M)
        END
   END HandleMyObject;
```

Listing 3: Record extension can also be applied to message records leading to a hierarchy of message types.

The dominant role of message records is also evident from systems such as MacOS, X11 or Windows where they appear as event records. In these systems, however, message records are expressed as non-extensible variant records (unions).

If efficiency rather than flexibility is crucial, there are further mechanisms available. In Oberon-2, which is supported by all commercial vendors, they include also type-bound procedures, which are similar to virtual functions in C++. A procedure Display, for example, can be bound to a type Line in the following way:

```
        PROCEDURE (L: Line) Display (F: Frame; x, y: INTEGER);
```

C++ introduces object-oriented programming via a special syntactic construct, the class, which is a textual bracket around an extensible structure definition and functions bound to this structure. Although message records and handlers would also be possible in principle, this technique is not practicable due to the missing type test operator.

In the typical C++ OOP-style with classes and virtual functions, code inheritance cannot be expressed explicitly as with Oberon's message handlers. Therefore, the language already contains several important inheritance relations including multiple inheritance and virtual base classes. These predefined mechanisms can, however, not compete with the flexibility of an explicitly programmed message handler. Generic forwarding of messages is for example not possible.

Exception handling

Most language designers now agree that I/O operations, processes, threads, semaphores and similar things should not be defined within the language since there are too many different concepts and none of them is appropriate for all applications. This does not mean that programmers should not use these concepts but that they should be provided by means of modules instead of language constructs. This idea is also applied to exception handling in Oberon, whereas in C++ a particular exception handling mechanism is already defined within the language.

Genericity

A program is called generic if it is not specific to a particular programming task. In a strongly typed programming language types are usually constant, i.e. specific. It is, however, also possible to think of program components such as procedures or classes which are parameterized with types. The most prominent examples of such generic programs are container classes (lists, sets or trees), that consist of elements of a given type.

C++ allows the usage of 'templates', i.e. building blocks which can be parameterized even with types in order to increase program reuse. Unfortunately, there is no better implementation technique known than expanding templates for all different argument combinations. Templates actually represent another kind of preprocessor, one that knows about the scoping rules of C++. Maintainance of expanded templates accross compilation units further complicates template implementation and usage. In current implementations this probem is mostly unsolved and frequent use of templates often leads to surprising code sizes due to unintended code duplication.

For this reasons, Oberon does not include a template mechanism within th language but delegates the task of expanding code fragments to the programmer. In principle, a template preprocessor would also be possible for Oberon, however, as a separate tool.

Overloading

One of the central design decisions of C++ was that it should be possible to define new data types that look exactly like built-in types. Consequently, it is necessary to allow user defined operators such as + or -. It is straight forward to extend this idea to overloading of functions as well.

In contrast to C++, one of Oberon's central design decisions was that imported objects should always be prefixed by the exporting module name inorder to ease reading of programs. This is in conflict with operator and function overloading. Therefore, Oberon uses overloaded operators and functions only for language defined types. This restriction also helps to guarantee that overloaded operators are similar enough to justify overloading and helps to reduce unintended introduction of inefficiencies. Please note also, that overloading, although an established mathematical concept, has its pitfalls. The interested reader might want to find out which one of the following two functions is called (if any) and what happens if one of them is removed.

```
void f(char*);
void f(int);
... f(0);
```

Summarizing, it can be stateted that the exceptional shortness of Oberon's language definition - less than 20 pages - does not origin in deficiencies of the expressivity of the language. Quite to the contrary, Oberon already contains some features which C++ programmers can only dream about. To mention just a few: modules, runtime type information and garbage collection. The latter is a necessary prerequisite for robust and reliable extensible software systems that will become more and more important in the future. It is hoped that the excellent educational Oberon implementations will be accompanied soon by equally well-implemented industrial programming tools to give also the practitioner a real alternative to C++.

Comparison between Oberon and C++

Criteria	Oberon	C++
type test	yes	no1
type BOOLEAN	yes	no1
modules	yes	no1
marking system-level programs	yes	no
defined initialization order	yes	no
garbage collection	yes	no
dynamic arrays	yes	no
runtime tests	yes	no
completly type safe	yes	no
preprocessor necessary	no	yes
exceptions	no	yes
templates	no	yes
overloading	no	yes
typing	explicit	implicit
precedence levels	4	17
language report (pages)	20	150

1 extensions are being discussed

Literatur

[1] R. Gericke: Wider den Schnickschnack; Oberon-System Teil 1: Anwendersicht; c't 2/94, p. 180 ff., Teil 2: Technische Einblicke; c't 3/94; p 240 ff.

[2] M. Reiser, N. Wirth: Programming in Oberon; Addison Wesley 1994.

[3] B. Stroustrup: C++ Programming Language; Addison Wesley, 1992.

[4] Ch. Kirsch: Entwicklertrauma: Markt~ubersicht: Tools f~ur die C-Entwicklung; iX 6/94; p 124 ff.

[5] B. Stroustrup: The Design and Evolution of C++; Addison Wesley 1994.

Remarks

Josef Templ received his Ph.D. in 1994 from ETH Zurich for his work on metaprogramming in Oberon.

Oberon can be downloaded via anonymous internet file transfer from ftp.inf.ethz.ch:/pub/Oberon

---

The ModulaTor Forum

Short report from JMLC'94

by G~unter Dotzel

The conference was well organized and the Modula-2 and Oberon-2 compiler product exhibition was very diversified.

The Oberon System V3 with gadgets looks very good and after Gutknecht's and J.L. Marais' talk, I'm now convinced that Gadgets is a good concept which already proved in many practical applications. ETH improved and completed documentation for V3/Gadgets recently.

(After we've completed the port of the Oberon System V4, ModulaWare will also port V3 and Gadgets to AXP under OpenVMS/OSF/Motif.)

I got positive feedback from many who listened to my talk about ModulaWare's M2 and O2 compilers for DEC Alpha AXP/OpenVMS. Many didn't know the most important architectural differences between the AXP and other RISC processors and what impact that has for compiler writers.

Even though it was the second last talk on Friday afternoon, there were still a lot of peoples listening. More than I'd have expected. Most attendees (together about 150) came from the University and there were only a few from industry. In this respect, the recent OberonDay'94 at ETHZ was better, because they achieved to attract more attendees from the (Swiss) industry.

Prof. Schulthess decided to have more talks about the use of Modula and Oberon in industry next time in order to attract more industry professionals. The next JMLC is scheduled for March 19 - 21, 1997 at Johannes Kepler University Linz, Austria.

I had the honour to talk to Wirth (he recovered very well from a recent heart-attack; maybe he has seen the ISO 10154 M2 DIS ;-), Gutknecht, Marais and Moessenboeck as well as many other VIPs at the final JMLC program committee party.

Last but not least:

The JMLC'94 conference proceedings are now available for DEM 90.- (540 pages, ISBN 3-89559-220-x). Most papers are either Oberon or OOP related.

You may send orders directly to Universit~atsverlag Ulm, Pf. 4204, D-89032 Ulm, fax ++49 731 42087.

Oberon CD-ROM Content

Content of the Oberon CD-ROM by Josef Templ (Addison Wesley, 1994, ISBN 3-89319-791-5):

Oberon documentation in RTF, WRI, TXT, MAC, UNX format.

Oberon System V4:

Docu,

Implementations: Windows, macII, PowerMac, Linux, Solaris 1 and 2, RS6000, Ultrix, irix4, irix5, hp700, amiga,

Examples: asciicod, vga, jay, sortdemo, redcode, fontedit, foldelem, class, find, labels, coco, kepler,

PO, PIO, OOP2 source from Oberon books

Oberon System V3:

Docu,

Impl.: DOS, Solaris 1 and 2, Linux,

Examples: asciicod, bartsimp, errmarks, outlines.

Demos for commercial Oberon implementations:

OM2, Mithril O2 and M2 for PC/DOS protected mode (ModulaWare)

H2O VAX/VMS Oberon V1.29 (ModulaWare)

The ModulaTor issues in PostScript up-to mdlt48 (Sep-94) plus PostScript to plain-text converter. (ModulaWare)

and more.

Press Release: New Product Announcement

XDS: OM2's ANSI C System

Erlangen 03-Oct-1994: ModulaWare now distributes OM2-XDS V2.x from XTech Ltd.

OM2-XDS is an Oberon-2 and Modula-2 compiler running on PC/DOS (386 and up) as well as all other popular computer platforms. OM2-XDS generates ANSI C code.

XDS' compiler front-end is based on OM2 and like OM2, XDS also comes with the standardised ISO Modula-2 10154 Library. The XDS library implementation is distributed in C-source code.

For more information about XDS, click here.

IMPRESSUM: The ModulaTor is an unrefereed journal. Technical papers are to be taken as working papers and personal rather than organizational statements. Items are printed at the discretion of the Editor based upon his judgement on the interest and relevancy to the readership. Letters, announcements, and other items of professional interest are selected on the same basis. Office of publication: The Editor of The ModulaTor is G~unter Dotzel; he can be reached by tel/fax: +33 492.81.30.99 or by mailto:gd@modulaware.com or mailto:modulaware@lycosmail.com In Germany contact: G~unter Dotzel, ModulaWare, Haselbachstr. 113, D-97653 Bischofsheim, Tel/Fax (+49) 9772 7101, mailto:infoD@modulaware.com In France contact: ModulaWare, La Chanenche, F-04340 Meolans Revel, mailto:info@modulaware.com

Most of the ModulaTor back-issues can be downloaded from http://www.modulaware.com/ in plain-ascii text and GNUzip-compressed PostScript format.

Home Site_index Contact Legal Buy_products OpenVMS_compiler Alpha_Oberon_System XDS_family DOS_compiler ModulaTor Bibliography Oberon[-2]_links Modula-2_links Effekta_onduleurs

webmaster@modulaware.com, 14-Nov-1998