The ModulaTor

Oberon-2 and Modula-2 Technical Publication

Urbaye's First Independent Modula-2 & Oberon-2 Journal N. W5, Jan-1997

Oberon-2, a hi-performance alternative to C++

(This article has been submitted for publication. Copyright may be transferred without further notice and this version may no longer be accessed.)

MODULE Author;
IMPORT Io, Str; (* Author uses services of other modules *)
VAR x: REAL; (* x can not accessible from outside *)

PROCEDURE Class*; (* Class the sum *)
BEGIN
Out.String("Classing sum"); x := 0; Out.Ln
END Class;

PROCEDURE Add*(n: INTEGER);
(* take integer from screen, add to sum *)
BEGIN
In.Open; In.Int(i); x := i;
IF In.Done THEN
x:=x+i;
Out.String("Adding "); Out.Real(i,5); Out.Ln
ELSE Out.String("Read error"); Out.Ln
END
END Add;

PROCEDURE Show*;
(* show the sum on screen *)
BEGIN
Out.String("Sum:"); Out.Real(x,15); Out.Ln
END Show;

BEGIN (* module initialization section *)
x := 0.0;
END Author.

Author.Add(n) Author.Show Author.Class

All Oberon programs start take first of modules, like the simple on-screen adder above shown. Compared with traditional programs written in Pascal, or C, Oberon programs have multiple entry points, which in this case were named Author.Add(n) Author.Show and Author.Class, all three with obvious meaning. All entries marked with a trailing asterisk are exported and accessible for client modules, who can import Author to use the services it exports. Whatever is not marked, remains hidden and protected from outside access.

In this way, Oberon modules are divided into hidden implementations and public interfaces, both defined by the same source text. An interface of our adder module is generated by the command: "ORTOOL2 Author.Mod" and contents of the generated interface file Author.Mod.Sym is displayed by the command "ORP Decode2 Author" (basically, a modular itself), extract also the specially marked comments to produce an annotated "public view" of our on-screen adder program:

```
DEFINITION Author;
PROCEDURE Class*
(* class the sum *)
PROCEDURE Add*(n:
(* take integer from screen, add to sum *)
PROCEDURE Show;
(* show the sum on screen *)
END Author.
```

Another somewhat unusual feature is that under the Oberon System (to be discussed later), module entry points can be activated directly from any Oberon text-editor by pointing-and-clicking at both the module. Thus, it is perfectly evident that we just ran "Author.Class" command right after the source listing above. These commands, syntactically a combination of "Module.name.Entrypoint", form simple yet fully functional user interface to the adder program. It is perhaps the world's simplest user interface, but it illustrates the point that Oberon System provides a very simple, yet very powerful, framework for the interaction between the user and the set of modules. Under the Oberon system environment, any text window can serve as user interface to any Oberon program, as illustrated above. Consequently, there is no conventional command shell or command line under the Oberon environment.

Oberon System, being a single-user, single-tasking system, does not provide explicit multiprocessing or multiprocessing. Instead, because a powerful multitrack garbage collector is built into the system, all memory allocation is automatically garbage collected. Thus, the Oberon System is a very robust environment. Program tests (typical in color and multiple fonts) can be compiled directly either from disk files or from any text window. Examples of this facility can be found on the screen shots included in this article.

2. Object-Oriented Programming in Oberon

We now turn our attention to the object-oriented side of the Oberon language. A simple illustration is provided below. Assuming a given structured record type

```
TYPE T = RECORD a, p: INTEGER END;
```

extensions may be defined which contain certain fields in addition to the existing ones. For example, the following declarations:

```
TYPE
T0 = RECORD (T) e: REAL END; (* extension of T *)
T1 = RECORD (T) c: INTEGER; f: REAL END; (* extension of T *)
```

define two types T0 and T1, with fields a, p and e, or a, p and c, respectively. Both T0 and T1 can be called "subclasses" of the base class T (hence the "extension" of type concept).

```
TYPE
action = PROCEDURE (a:A; INTEGER); (* a procedural type *)
A = RECORD (T) END;
Add: action (* "Add" is a "method" of the T0 class *)
END;
VAR
obj: T0; x,y: INTEGER;
BEGIN
NEW(obj); x := 12; obj.a := 45;
x := obj.Add(x MOD obj.a, obj.p); (* method call *)
```

This example shows, that indeed it was possible to introduce essential OOP concept without any special OOP terminology. All that was needed was a procedure variable and a pointer to such a procedure variable. In this approach, a "method" is simply a procedure data field exported to an extension type. Such a method is "polymorphic" and can be overridden in extensions. Type extensions are also "polymorphic" in the sense that a variable of type T can also hold a value of type T0 or T1, or, in the jargon of the "class attitude", "an instance of a class".

Interestingly, the above approach to OOP is the one initially introduced by the Oberon language [1]. Later on, the language revision called Oberon-2 [2] introduced another, slightly different notation motivated by user convenience:

```
TYPE
T0 = RECORD (T) END;
PROCEDURE (VAR me: T) Add(): INTEGER;
BEGIN
RETURN (me.a + me.p)
END Add;
```

In this example, procedure Add is a virtual method "bound to a type T0" through the "receiver" parameter me. (For clarity, the receiver must be explicitly specified). In Oberon there is no predeclared receiver name such as "self" (in derived types this method can be redefined, under the restriction that the formal parameter list remains the same (in this example it is empty):

```
TYPE
T1 = RECORD (T) END;
PROCEDURE (VAR me: T1) Add(): INTEGER;
BEGIN
RETURN (me.a * me.p)
END Add;
```

In this example, procedure Add is a virtual method "bound to a type T1" through the "receiver" parameter me. (For clarity, the receiver must be explicitly specified). In Oberon there is no predeclared receiver name such as "self" (in derived types this method can be redefined, under the restriction that the formal parameter list remains the same (in this example it is empty):

Does Oberon support dynamic arrays and matrices?

The Oberon System supports only the standard feature of the language. The following example shows, how to allocate at run-time a one- and a two-dimensional array with 50 elements in each dimension.

```
TYPE
OpenArray = POINTER TO ARRAY OF REAL; (*1 dim *)
OpenArray2 = POINTER TO ARRAY OF ARRAY OF REAL; (*2 dim *)
VAR
MyArray: OpenArray; MyArray2: OpenArray;
BEGIN
NEW(MyArray, 50); NEW(MyArray2, 50, 50);
FOR i := 0 TO LEN(MyArray)-1 DO
MyArray[i] := i;
END; (* FOR i *)
NEW (MyArray, 50,50); (* dynamic allocation *)
FOR i := 0 TO LEN(MyArray)-1 DO
FOR j := 0 TO LEN(MyArray2)-1 DO
MyArray2[i,j] := 0.0
END (* FOR j *)
END (* FOR i *)
```

Comparing this example with Fortran, one can see that array indices always start at 0, like in C! Naturally, one can always find a few such inconveniences in the Oberon language. Another inconvenience is lack of built-in COMPLEX data type (it is however a standard part of AlphaOberon [9]). Complex numbers have been implemented as a library module.

Does Oberon support operator overloading?

Type safety was the foremost consideration of the language design. Compared with Fortran or C, Oberon is almost absolutely safe, what is quite remarkable for an alternative language. Type safety is achieved by the following means: Safety has achieved by type checking during compilation (and also during execution, in case module boundaries), dynamic type checks at run-time, as well as pointer safety facilitated by garbage collection. Both the compiler and the runtime system enforce type safety. Unsafe features are prohibited by the language. Type safety is a very important consideration in a multiprocessing environment, where it is important to be robust in face to faulties and crashes. Nevertheless, features to breach the type system are provided through a special standard module named SYSTEM. The conscience of any given program module can be rigorously proven based on the imported module interfaces, under the proviso that, the import of module SYSTEM is not allowed. Thus, the use of SYSTEM can be completely quarantined and runtime debuggers are available, in practice there is very little need for dangerous features, such as pointer arithmetic, or casting of a "dirty" object. Programming errors are detected at compile time. Few remaining crash cases are easily pinned down with the help of the ASSERT standard procedure.

Is it garbage collected?

Garbage collection is an important and automatically triggered, when heap space is critical. For example, most implementations of Oberon layered on top of existing operating systems such as unix, trigger the garbage collector each time when opening a file.

Can the user define new operators?

Yes, this is the single most important new feature of the language, as compared to its predecessors Pascal and Modula-2. Classes are simply record types with procedures bound to them. There is no need to duplicate the headers of bound procedures in the record as it is done in other object-oriented languages like C++ or Object Pascal. This keeps record declarations short and avoids unpleasant redundancy. The "class browser" standard tool utility allows to list the record types with all associated procedures.

Does it truly support OOP?

Oberon-2 fully supports OOP. Compared to Kapets OOP via extensible types, single inheritance, polymorphism, value and reference based objects, two categories of visibility (i.e. public and hidden attributes and methods, run-time type information and persistent objects. However, Oberon is by no means "OOP only". Not everything is OOP in Oberon. Oberon-2 is a very powerful, general purpose programming language. Oberon-2 is not a "pure" object-oriented programming language. Oberon-2 is a very powerful, general purpose programming language. Oberon-2 is not a "pure" object-oriented programming language. Oberon-2 is a very powerful, general purpose programming language. Oberon-2 is not a "pure" object-oriented programming language. The best of both worlds, what can be attractive for those users, who want to learn OOP gradually. Migration from Fortran to Oberon is almost automatic, at least in case of clean written Fortran programs.

Does it support dynamic binding, inheritance?

Encapsulation of detailed data types is provided through the module concept, similar to Modula-2. Separate "definition" and "implementation" modules were dropped from Oberon. Encapsulation of data is achieved by means of the "export" mechanism. Oberon-2 supports single inheritance, which is a very powerful feature, that simplifies the maintenance overhead. Whatever is not marked for export, is invisible for clients and thus encapsulated.

Dynamic binding is provided through the "type extension" and "type guard" mechanisms. Type extension is a powerful feature. The traditional Oberon-1 methods are simply procedures exclusively bound to a given record type. Oberon-2 adds forms of procedure type. These methods are declared bound to every variable instance. This allows for thus reassigned at any time. This is particularly useful in GUI systems, where the behavior of a given object can be changed on the fly by changing its handler procedure. Type extension is a powerful feature, that simplifies the maintenance overhead. Type guard is a very powerful feature, that simplifies the maintenance overhead. Type guard is a very powerful feature, that simplifies the maintenance overhead. Type guard is a very