The Oberon Programming Language

Source Code Accompanies This Article. Download It Now.

- oberon.asc

Oberon, a general-purpose, object-oriented programming language that evolved from Pascal and Modula-2, has been implemented for DOS, Windows, Amiga, Mac, and UNIX.

January 01, 1994
URL:http://www.drdobbs.com/architecture-and-design/the-oberon-programming-language/184409405

Josef is associated with ETH Zurich and can be contacted at jt@swe.uni-linz.ac.at.

Oberon is a general-purpose programming language that evolved from Pascal and Modula-2. This evolution included improvements such as garbage collection, a streamlined module concept, numeric type inclusion, and null-terminated strings, but Oberon's principal unique feature is the concept of type extension. Type extension makes Oberon an object-oriented language. However, Oberon's approach to object-orientation differs considerably from that of other extensions of Pascal or Modula-2. Oberon implementations for DOS, Windows, Amiga, Mac, and UNIX are available via anonymous ftp from ftp.inf.ethz.ch (129.132.101.33) subdirectorypub/Oberon.

Modules

The most important difference between Pascal and Modula-2 is the decomposition of programs into modules. In Oberon, a program is an extensible set of modules; in other words, there is no main module. The unit of program execution in Oberon is the "command," an exported, parameterless procedure. Thus, a module in Oberon is:

- The construct for expressing data abstraction by means of an import/export mechanism, like in Modula-2.
- The unit of compilation, including type checking across module boundaries.
- The unit of program extension. A module is supposed to be loaded on demand whenever it is used first.
- A container of commands which may be activated from the operating environment.

Example 1 illustrates a typical Oberon module. The import list of *M* lists all modules which should be accessible inside *M.* Module *M* is said to be a "client" of *M1* and *MyModule.* A client can only access those objects of an imported module which are exported. The asterisk (*) is used to signal export of an object; for example, *T** means that type *T* is to be exported. In contrast to Modula-2, Oberon unifies the definition and implementation modules by means of the export mark, which also allows you to selectively export record fields. In addition, the import list lets you rename the imported module to enable simple substitution of one module by another. In Example 1, *MyModule* is imported under the alias *M2.* Imported objects such as *M1.P* are always qualified with the exporting module in order to avoid name clashes and to increase program readability and maintainability.

Infinite Heap

Inappropriate deallocation of dynamic storage is a well-known source of catastrophic program failures. In the context of nonextensible applications, such as statically linked programs, this problem can (in theory) be mastered by a careful programmer; for extensible programs, it cannot! There is always the possibility that a module may be added to a program later on; such an extension can introduce additional references to objects of which the implementor of the core modules is unaware. Therefore, Oberon neither requires nor allows explicit deallocation of dynamic storage. Instead, it assumes a conceptually "infinite heap." As in Lisp or Smalltalk, this concept can be implemented on today's finite hardware by a garbage collector which knows about the internal structure of objects and the roots where objects are anchored. Unused objects can be identified and deallocated automatically and safely. This facility makes Oberon programs very reliable, significantly decreasing debugging time. Automatic garbage collection is possible in Oberon because Oberon replaces typing loopholes such as records with variant parts with the type-safe concept of "record extension."

Basic Types

Oberon's basic types are fairly familiar: BOOLEAN, CHAR, SET, SHORTINT, INTEGER, LONGINT, REAL, and LONG-REAL. Between the numeric types, there is a type-inclusion relation: The larger type includes the smaller ones.

With LONGREAL, REAL, LONGINT, INTEGER, and SHORTINT, assignment from a smaller to a larger type is allowed. Operations between different numeric types yield the larger type as result.

SET is a rather crude approximation of the mathematical set concept. It denotes the power set of the integers between 0 and MAX(SET), which is an implementation-dependent constant (typically, 31).

Type Constructors

In addition to the basic types, Oberon allows construction of user-defined types by means of the type constructors ARRAY, RECORD, and POINTER. Unlike standard Pascal, however, Oberon offers the type constructor PROCEDURE, which defines procedure types. For example, *TYPE Poly2 = PROCEDURE (x0, x1, x2: REAL): REAL;* introduces a procedure type denoting quadratic polynoms. Variables of type *Poly2* can have as values procedures with appropriate parameters. Procedure variables introduce a level of indirection for procedure calls and can be used to express dynamic binding--a prerequisite for object-oriented programming.

On the other hand, *TYPE String = ARRAY 32 OF CHAR;* defines a new type *String* as an array of 32-character elements indexed from 0 to 31. In contrast to Pascal, the lower bound is always 0, for several reasons:

1. Open arrays already start at index 0 in Modula-2.
2. The MOD-operator (positive remainder) yields results including 0. Lower bounds at 0 fit perfectly, for example, for an implementation of a cyclic buffer.
3. There is a nice invariant when iterating over an array.The control variable contains the index of the next element to visit as well as the number of already visited elements.

Fixing the lower bound at 0 practically eliminates off-by-one errors.

In contrast to Pascal, Oberon lets you declare formal parameters as open arrays without specifying the number of elements. Such procedures can be called with arrays of arbitrary length as actual parameters. Unlike Modula-2, Oberon allows you to specify multidimensional open arrays.

String literals are compatible with arrays of characters. String assignment and comparison are defined within the language, based on the assumption that strings are always null-terminated.

Record-type constructors such as *TYPE T = RECORD x: CHAR; y: INTEGER END*, are similar to those in Pascal except that variant records have been replaced by record-type extension. This means that a new record type can be defined as an extension of an existing one. For example, in *TYPE T1 = RECORD (T) z: REAL END;*, type *T*1 is said to be a direct extension of type *T,* which is a direct base type of *T*1. The extended type inherits all fields of the base type--everything that can be done with the base type can also be done with the extended type but not vice versa. Therefore, Oberon allows you to assign variables of an extended type to variables of the base type.

In contrast to Pascal's variant records, record extension is an open-ended construct; an extended record may well be defined in a module other than the corresponding base type. Surprisingly, record-type extension suffices to make Oberon an object-oriented language.

The code *TYPE P = POINTER TO T;* is similar to Pascal's *^T. POINTER TO T* constructs a new pointer type that denotes references to variables of type *T.* Pointer types inherit the compatibility relations of their base types--a pointer to an extended record is regarded as an extension of a pointer to the base record. Assigning a pointer variable of an extended type to a pointer variable of the base type introduces the notion of "dynamic type."

Static and Dynamic Type

The static type of a variable in Oberon is the type specified together with the declaration of the variable. The dynamic type of a variable is the type the variable assumes at run time. The dynamic type can only be an extension of the static type.

Consider, for example, two pointer variables *v*1: *T*1 and *v: T,* where *T*1 is an extension of *T.* If *v*1 holds a reference to a variable of type *T*1--that is, by a preceding NEW(*v*1)--the assignment *v* := *v*1 assigns a reference to a variable of an extended type to a variable of the base type. After the assignment, *v*'s dynamic type is *T*1, the static type remains *T.* This rule also applies to passing extended records by reference (that is, as VAR-parameters). Only base-type fields can be accessed via *v,* but the extended fields of *T*1 still exist.

Reverse Assignment

The reverse assignment *v*1 := *v* is only meaningful if the dynamic type of *v*1 is at least *T*1, the static type of *v*1. In Oberon, this property can be tested by the type-test operator "IS". The expression *v* IS *T*1 yields True if, and only if, the dynamic type of variable *v* is at least *T*1. The assertion that a variable is of a given dynamic type can be expressed by the type guard, written as *v*(*T*1). This asserts (at run time) that variable *v* is at least of dynamic type *T*1 and allows access to the extended fields of *T*1. The reverse assignment can be written as *IF v IS T1 THEN v1 := v(T1) ELSE _ END.* If the same type guard has to be applied several times in a statement sequence, the With statement can be used as a regional type guard; see Example 2.

Object-Oriented Programming in Oberon

The paradigm of object-oriented programming is based on the assumption that objects communicate with each other by sending messages. In the real world, messages are often represented as letters, notes, videos, e-mail, and the like--that is, as objects themselves. A powerful OOP style, therefore, must let you represent messages as explicit objects rather than as parameters of procedure calls (SIMULA-style OOP). Oberon can use records to represent both objects and messages and record extension to create hierarchies of both object and message types. In Example 3, the types *CopyMsg* and *ConsumeMsg* are derived from the base type *ObjMsg.*

To respond to this sort of messages, Oberon expresses the behavior of an object using the "message handler" (or handler) procedure. Example 4 is an example of a simple handler. There are two parameters, one describing the receiver of the message and the other representing the message sent to this object. The message is passed as a record VAR-parameter, allowing use of type tests to identify messages. Messages of unknown type are usually ignored. Error messages such as Smalltalk's message-not-understood are not appropriate here since--in analogy with the real world--it is perfectly legal to ignore a message without committing suicide.

The handler is usually bound to an object via a *record* field of procedure type; see the simple object model in Example 5(a). New object types are introduced as extensions of the base type, as in Example 5(b). To generate a new object, you create a new variable and assign a handler to the variable's *handle* field; see Example 5(c). To send a message, you set up a message record and call the handler of the receiving object with the receiver and the message as parameter; see Example 5(d). Looking at the possible variations of the fundamental message-send pattern *o.handle(o, m)* reveals many important OOP concepts.

Inheritance

An object can inherit behavior by simply activating another handler. In Example 6, for instance, *MyHandle,* the handler of *MyObject,* inherits behavior by calling *Objects.Handle* for all messages that are not specially handled. Instead of an object, a module is used to select a handler, and static binding takes place. The receiver and message remain unchanged. This technique of expressing inheritance has several properties:

- Inheritance is programmed explicitly, rather than predefined by the language.
- A wide variety of inheritance relations is possible.
- No special syntax or semantics in the language are required.
- Subtyping and subclassing are disentangled.

Delegation

In general, delegation means inheriting from an object. In the context of our message-sending pattern, delegation means using different objects for selecting a handler and acting as receiver. In Example 6, exchanging the ELSE branch with *ELSE traits.handle(O, M)* means that object *O* delegates all unknown messages to object traits, which is supposed to provide suitable standard behavior. The receiver and the message remain unchanged.

Forwarding

An object can also forward messages to other objects. The most prominent examples are container objects, which handle several messages in a special way and forward all others to their components. Again in Example 6, an ELSE branch of the form *ELSE x.handle(x, M)* means that object O forwards all unknown messages to object *x. M* remains unchanged; the receiver of the new message becomes *x.*

Broadcasting

An important application of forwarding is broadcasting a message to a group of receivers, as in Example 7. Typical applications of broadcasting are container objects that consist of more than one component. In the Oberon system, broadcasting is also used heavily for sending messages to all visible objects on the screen in order to provide consistency between the model and the view. Note that the Broadcast procedure is a generic iteration construct that is independent of a particular message type--that is, it works for all messages, even those introduced years later in additional modules.

Through forwarding, messages can arrive at an object via different paths at different times. To handle this, an object needs information about the context of a received message. To this end, Oberon System 3 uses time stamping of messages to detect multiple arrivals and a dynamic link that points back to the sender of a message.

Efficiency Considerations

OOP based on message records and handlers seems, at first glance, rather inefficient and inconvenient. In practice, however, efficiency is not a big problem since a type test can be carried out in a constant, very short time (two loads plus one compare). It's also possible to speed up message identification by grouping messages, either using type extension or by introducing an explicit tag. Message forwarding is very efficient, and is independent of message-record size. Furthermore, specialized procedure variables can be introduced within an object to reduce the message-dispatching effort to the call of a procedure variable.

Event records are a similar technique used in contemporary operating or windowing systems. They are expressed by variant records (unions in C) that are not open ended and type safe. However, inefficiencies in the Macintosh or Windows operating system, for example, do not originate from the use of event records. Oberon's message records are "cultivated" event records.

Run-Time Environment

The Oberon programming language has been developed together with an operating environment, the Oberon system. The language, however, makes very few assumptions about the environment. If not available from a given operating system, dynamic module loading, command invocation, and automatic garbage collection can be introduced by a small run-time system. In principle, it is possible to forego dynamic loading and to create traditional, statically linked applications, but this is anachronistic for the challenges of today's software systems. Automatic garbage collection is indispensable for reliable, extensible software systems and is probably the biggest culture clash between Oberon and other compiled languages such as Pascal, C, or C++.

References

Wirth N. and M. Reiser. *Programming in Oberon: Steps Beyond Pascal and Modula-2.* Reading, MA: Addison-Wesley, 1992.

Example 1: Typical Oberon module.

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

TYPE
  T* = RECORD
    f1*: INTEGER;
    f2: ARRAY 32 OF CHAR
  END ;
```

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

END M.
```

Example 2: Reverse assignment.

```
WITH v: T1 DO
  v treated as being declared
with static
  type T1 in this statement
sequence
END
```

Example 3: The types CopyMsg and ConsumeMsg are derived from the base type ObjMsg.

```
TYPE
  Object = POINTER TO ObjDesc;
  ObjDesc = RECORD ... END ;

  ObjMsg = RECORD END ;

  CopyMsg = RECORD(ObjMsg)
    deep: BOOLEAN; cpy: Object
  END ;

  ConsumeMsg = RECORD(ObjMsg)
    obj: Object; x, y: INTEGER
  END ;
```

Example 4: A simple Oberon handler.

```
PROCEDURE Handle (O: Object; VAR M: ObjMsg);
BEGIN
  IF M IS CopyMsg THEN handle copy message
  ELSIF M IS ... handle further message types
  ELSE ignore
  END
END Handle;
```

Example 5: The handler is usually bound to an object via a record field of procedure type.

```
(a)
TYPE   Object = POINTER TO ObjDesc;
  ObjMsg = RECORD END ;
  Handler = PROCEDURE (O: Object; VAR M:
ObjMsg);
  ObjDesc = RECORD
    handle: Handler
  END ;

(b)
TYPE
  MyObject = POINTER TO MyObjDesc;
  MyObjDesc = RECORD (Object)
    extended fields
  END ;

(c)
VAR o: MyObject;
NEW(o); o.handle := MyHandle;

(d)
VAR m: CopyMsg;
m.deep := TRUE; m.obj := NIL;
o.handle(o, m);
```

Example 6: MyHandle, the handler of MyObject, inherits behavior by calling Objects.Handle.

```
PROCEDURE MyHandle (O: Object; VAR M: ObjMsg);
BEGIN
  WITH O: MyObject DO
    IF M IS CopyMsg THEN handle copy message
    ELSIF M IS ... handle further message types
    ELSE Objects.Handle(O, M)
    END
  END
END MyHandle;
```

Example 7: Broadcasting a message to a group of receivers.

```
PROCEDURE Broadcast(VAR M: ObjMsg);
  VAR o: Object;
```

```
BEGIN o := firstObj;
  WHILE o _ NIL DO o.handle(o, M); o := nextObj END
END Broadcast;
```