191

A Comparison of Eiffel, C++ and Oberon-2

Audrey Mayes and Mary Buchanan

March 25, 1994

Contents

1 Introduction

The languages Oberon-2 [1], Eiffel [2, 3] and C++ [4, 5, 6] allow an object-oriented style of program to be implemented. Oberon-2 and C++ also support other styles of programming.

All the languages are available on the SPARC workstations. Information for accessing Oberon-2 and Eiffel can be found in [7] and [8].

This report presents a comparison of the object-oriented features of the three languages. A knowledge of object-oriented concepts is assumed.

The three languages use different terms to define the same constructs. For this reason the comparison begins by discussing the terminology used. The comparison continues by considering the essential features of an object-oriented language. These features are:

- encapsulation and information hiding,
- type extension or specialisation which is usually known as inheritance,
- polymorphism and dynamic binding,
- object identity.

Inheritance is the only feature unique to object-oriented software.

Other language features are then compared. The chosen areas of comparison are:

- units of modularity,
- types provided,
- redeclaration of features or procedures,
- type checking and type conformance rules,
- accessing the dynamic type of objects,
- system execution,

2 Terminology

The three languages use different terminology. C++ and Eiffel both refer to classes and objects. The term class is used, in these languages, to mean the language construct used to define abstract data types. The Oberon-2 term for a class is either a record type with procedure variables or type-bound procedures or a pointer to such a record type. The rationale for using the term 'record' is that programmers can readily understand the concept [9]. All three languages use the term object to mean instances of a 'class'. Oberon also uses the term 'variable of the type' with the same meaning. The terms class and object are used in the subsequent discussion.

Methods are the means provided for accessing the state of an object or asking the object to perform a computation. Eiffel and C++ both provide one mechanism for implementing methods. These are the routines that are part of the class. Oberon-2 provides three different mechanisms. The mechanisms [7] are type-bound procedures, procedure types and message records. Type-bound procedures are very similar to the Eiffel and C++ implementation of methods. Procedure types provide similar functionality to type-bound procedures but are more complex to implement. The third mechanism, the use of message records, is not documented in either C++ or Eiffel. Message records provide greater flexibility than type-bound procedures but are not type checked.

Another difference in terminology relates to the formation of new classes from existing ones by extension. Oberon-2 uses the term 'type extension'. The other two languages use the more usual term 'inheritance'.

The final difference in terminology which is of importance to this report, is in the way the parts of an object are named. Oberon-2 refers to the fields and procedures of an object. C++ uses the term member to cover all parts of the object. Eiffel refers to all parts as features. The features can be either attributes or routines. The Eiffel terminology is used for the following discussion.

3 Encapsulation and information hiding

Encapsulation, or abstract data typing, is the grouping of data with the operations that can be applied to it. Classes can be considered to be implementations of abstract data types [2, 10]. Access to the data and operations encapsulated in a class can be controlled by the programmer which gives information hiding. The parts of a class which are exported, not hidden, form the interface to that class.

The C++ pre-processor allows a programmer to declare the interface in a separate file to the implementation of the class. The interface can be declared in a header file which must contain details of the data structure. It is also possible to declare the interface and implementation in one file.

Eiffel and Oberon-2 classes must be declared with the interface and implementation in one file. The use of one file is said to be quicker for the programmer and easier for the compiler which does not have to check consistency between the interface definition and the implementation modules. Another effect of using one file for both interface and implementation is that the interface can be altered simply by changing the export status of features. This does not encourage the completion of the design of a system before implementation begins. The interface of a module is bound to the implementation which makes it more difficult to change the data structures used to store the objects.

Oberon, C++ and Eiffel v2.3 apply the same general rule to govern information hiding. They declare that all parts of the class will be hidden unless declared specifically as exported. This rule adds to the security of code. However, Eiffel v3.0 has changed this rule. The default, in this version of Eiffel, is that features are available to all classes unless access is explicitly restricted [3].

The detail of export rules vary. In Oberon-2, two grades of export can be specified by the programmer. These are read-only export and read/write export. Eiffel automatically ensures that exported attributes are read only and exported routines are executable. C++ provides read/write export only but provides public export, giving all classes access to the feature, and protected export which allows only derived classes to access the feature.

A class in Oberon-2 has only one interface. All clients of a class are treated equally. There is no special arrangement for a class inheriting from another class. This means that it is impossible for a derived class to break the encapsulation of its parent class.

Both C++ and Eiffel allow classes to have several interfaces. These interfaces are created by using a number of facilities.

- General access by any class

  Client classes are allowed access to any features labelled as public in C++, or listed as exported in Eiffel v2.3, or exported to class ANY in Eiffel v3.0. All Eiffel classes are descended from class ANY so gain access to those features exported to class ANY.

- Access by some classes only

  In C++, classes or functions can be declared as friends of the class being defined. These friend classes or functions can access any feature of the C++ class whether public, protected or private (hidden). Eiffel allows a programmer to export individual features of a class to another class or group of classes. Any feature from the named class can access the individual feature. This is known as selective export.

- Descendant access

  In Eiffel descendant classes gain access to all features of the parent class. Class invariants are inherited to help prevent the misuse of inheritance by breaking the encapsulation. C++ provides greater control over descendant classes access. Descendant classes can access the public or protected fields of a class but not the private fields.

Eiffel and C++ provide encapsulation and information hiding via the class construct but also allow means of breaking the information hiding. C++ also provides the *struct* construct which can be used to implement classes. All features of a class declared as a struct are public so this method of implementing a class does not provide the information hiding required by an object-oriented language. All future references to C++ classes refer to those implemented using the class construct.

4 Inheritance

Inheritance provides the means to build new classes based on existing ones. Oberon-2 has a strict interpretation of inheritance. Type-bound procedures can be redefined but the formal parameter list remains the same. A derived class must export all the features exported by the base class. The restrictions on the redefinition of procedures and export list enable the subtype relationship to hold for all derived classes. Some authors [11, 12] suggest that subtyping should be the only interpretation of inheritance, at least during the analysis and design phases of development.

Both Eiffel and C++ allow multiple inheritance. This means that classes can have more than one parent class. Eiffel permits the same class to be inherited repeatedly but C++ insists that a class can only be named once in its list of inherited classes. The languages both have mechanisms for resolving any name clashes that occur due to multiple inheritance.

It is possible in both C++ and Eiffel to change the export status of inherited features. C++ classes can be inherited using the keyword *public*. The export status of the inherited members remains the same as in the superclass. If the keyword, *public*, is not specified, another keyword *private* is assumed which results in all the features being private in the derived class. Eiffel allows the export status of inherited features to be changed from secret to public or public to secret. The ability to change the export status of inherited features is provided to increase reusability but means that a subtype relationship may not hold for derived classes in C++ and Eiffel. This has consequences for the type checking systems which are discussed in section 10.

The ability to change the implementation of inherited features is discussed in section 9.

5 Polymorphism and dynamic binding

Polymorphism in this context is defined as the ability to use the same procedure name to call different code depending on the class of the receiving object. Many classes can then respond to the same message, *print* for example. When polymorphism is combined with dynamic binding the correct code is chosen at run time.

All three languages have features which permit polymorphism. The binding of operations to classes permits any number of classes to define a print procedure which is uniquely identifiable. Redefinition of these class bound procedures allows subclasses to declare their own version of the code which is suitable for their instances.

The three languages approach the binding of code to procedure calls in different ways. In Eiffel, all procedure calls are bound dynamically. C++ and Oberon allow the programmer to chose dynamic or static binding because static binding results in greater run-time efficiency. Static binding is the normal situation in C++. In order for C++ procedures to be bound dynamically, they must be declared as virtual. In Oberon-2, type-bound procedures and procedure variables are bound dynamically but procedures which do not form part of an object are bound statically.

6 Object identity

Object identity ensures that it is possible to have several instances of classes which are identical apart from a system generated identifier. All the languages under comparison provide this feature. It is worth pointing out that many other languages also permit several variables to hold the same value. It is mainly in database environments that it is not possible to have many copies of identical variables. The user of relational databases must provide identifiers by which the copies can be distinguished.

Object identity means that all objects are unique. In this discussion, objects are considered to be declared as references to structures. The fields of two objects can contain identical information. These two facts give rise to three different interpretations of the term equality [2, 10]. Figure 1 uses instances of the class Person to demonstrate these three definitions of equality. The first interpretation is that two objects have the same unique identifier and are therefore the same object. This is known as the *identity predicate*. This situation would arise after the assignment `p1 := p2`.

The second possibility is that the fields of the objects, which may be pointers, contain the same values. This is known as *shallow equal*. This situation would arise after assignment statements such as

```
        p1.name := n1;
        p3.name := n1;
        p1.address := a1;
        p3.address := a1;
        p1.telNo := 4763;
        p3.telNo := 4763;
```

The third and final definition of equality is that, after following the pointers until a value is reached, the values of the fields of the objects are identical. This is known as *deep equal*. This situation would arise after a series of assignment statements such as

```
n1 := '' Mary'';
n2 := '' Mary'';
p1.name := n1;
```

```
p3.name := n2;
```

and so on for the other fields.

The languages differ in their support for these three concepts of equality. In all three languages being compared it is possible to compare for the identity predicate using the languages usual equality test. For example, in Oberon-2:

```
p1 = p2
```

tests the pointer values and returns a boolean value.

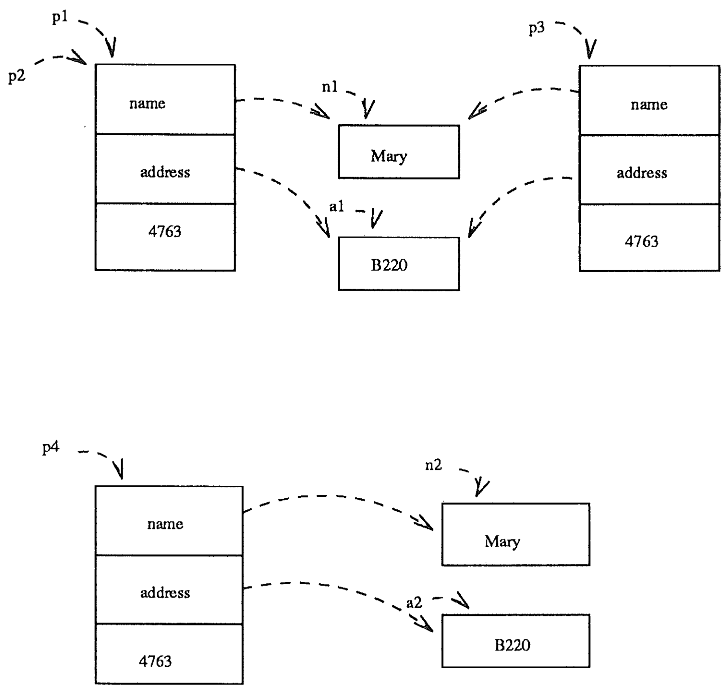
Eiffel is the only one of the three languages documented as supporting the other forms of equal. The predefined routines *equal* and *deep_equal* test for shallow equality and deep equality respectively. The programmer must implement routines to test for these other forms of equality in C++ and Oberon.

Similar possibilities arise when considering the copy operation. Copies can be either shallow or deep. Shallow copies result in field by field copies being produced. Any fields in the original object which contain references contain the same reference in the copy. This can be seen by referring to figure 1, a shallow copy of object p1 would produce p3. Deep copies result in new copies of every sub-object also being produced. The new copy does not share any fields with the original. In figure 1, object p4 is a deep copy of p1.

Again the languages differ in their support for copying. Oberon-2 leaves it up to the programmer to write copy procedures for objects. Eiffel provides a predefined routine *Clone* which provides shallow copy and deep copy routines. C++ expects the programmer to implement full copy semantics to ensure that a deep copy is made and that there is no shared data when an assignment is made. This is necessary because of the way C++ deals with formal parameters. Formal parameters are considered to be local variables and so are created on entry to a function and destroyed on exit from the function. Actual variables are assigned to the formal parameters at runtime so any shared fields would be destroyed on exiting the procedure.

7 Units of modularity

Units of modularity are the parts from which a system is assembled. Each part has a well defined interface. Classes are the basic unit of construction in object-oriented software production. In Eiffel the class construct is also the unit of modularity. This means that there must be one class per file and one file per class. Both Oberon-2 and C++ allow more than one class per file. This allows closely related classes to be declared in the same file. Oberon-2 allows the classes to be hidden so that they can only be used within the module. The files are the unit of modularity.



p1 and p2 point at the same object. This is the identity predicate.

p3 and p1 have fields with the same value. They are shallow equal because the
non pointer fields contain the same value,
the name fields point to the same object, n1,
and the address fields point to the same object,a1.

p4 and p1 have fields which point to different objects whose fields contain the same values,
and non pointer fields which contain the same value.
They are deep equal but not shallow equal.

Figure 1: Three Interpretations of Equality

In C++ [6], it is possible to declare one class within another class to produce a nested class. This nested class is in the scope of the enclosing class and must be accessed via the enclosing class. The declaration of a nested class does not mean that the enclosing class contains an instance of the class.

8 Types provided

Oberon-2, Eiffel and C++ all provide simple types, such as integer, real and char, and ways of implementing structured types. The support for generic types, enumerated types and subranges varies.

- Simple types

  Oberon-2, Eiffel and C++ all implement simple types as instances of the type not as pointers to instances of the type. All provide simple types for character, integer and real variables. Oberon-2 and C++ provide two sizes of real numbers and three sizes of integers. C++ is the only one of the languages not to provide a type boolean but use the integers 1 and 0 to represent true and false in boolean expressions.

- Structured types

  Classes provide implementations of structured types. Objects are instances of Class types. It is possible in all three languages to provide objects as either pointers or actual instances of the class. The mechanisms for providing the different types of objects vary.

  In Eiffel, either the class definition or the client module controls which type of object is produced. The default behaviour is that the declaration of an instance results in the allocation of a pointer. In order to make the pointer reference an actual instance, a *Create* procedure must be invoked on the pointer or the pointer can be assigned to another instance. If an actual instance of a class is required when the declaration is made, the client module can declare the instance as **expanded**. For example, an actual instance of type person could be declared as

  ```
  p : expanded Person;
  ```

  Alternatively the class declaration could begin with the keyword **expanded**, in which case all instances of the class will automatically be actual objects and not pointers.

  In the C++ language it is the client module which controls the production of objects. The programmer can declare either an instance or a pointer to an instance of a class.

In Oberon-2, a class can be declared as a record with attached procedures and/or as a pointer to a record structure with attached procedures. The programmer declares instances as appropriate.

In all the languages it is recommended that objects are pointers not actual instances of the class.

Arrays are also structured types. C++ and Oberon-2 provide arrays as basic language elements whereas Eiffel uses the generic type mechanism.

- Generic types

  Generic types are used to define data structures such as lists and trees. The abstract structure of lists etc. is common to all instances of the type. Many operations on these types, for example add and delete, do not depend on the types of the actual elements contained. It is these operations that are specified in the generic type. A restricted form of structural type equivalence is needed to provide type checking on generic types. Generic types are implemented as parameterized classes. The parameters for these classes represent types. Actual types, such as lists of integers, are produced by providing actual parameters for the formal parameters.

  Eiffel supports genericity. It is possible to implement both unconstrained genericity, where any type of class can be used as an actual parameter for the type, and constrained genericity where only specified types or their descendants may be used.

  Ansi standard implementations of C++ [6] provide templates as a means of implementing generic types. Other implementations provide a package, generic.h, which enables generic types to be declared [4].

  Oberon provides the type SET but this type can only be used with integers. It is claimed that Oberon-2 can be used to implement other generic types [1]. The suggested implementation of a FIFO queue relies on the "generic" elements being subtypes of the declared type. In the example, the declared type of element is an empty record, of type Node. All possible elements to be included in the list must be descended from this type, Node. The type conformance rules ensure that all these nodes will be compatible with the FIFO queue. Any instance of this type could be completely heterogeneous. In systems where it is desirable to declare more than one queue, each of which must contain specific types only, the programmer must use type guards. The type guards could be used in one of several ways. Two methods are:

  - use a type guard every time an element is added to the queue or removed from it.
  - implement the FIFO using type-bound procedures and redefine each procedure to include a type guard.

- Enumerated types

  Neither Oberon-2 nor Eiffel provide the facilities required to declare enumerated types. They were omitted from Oberon-2 because *"they defy extensibility over module boundaries"* and had been observed to have *"led to a type explosion that contributed ... to program verbosity"* [13]. It is possible to implement enumerated types in Oberon-2 by using numeric constants or arrays. Meyer [2] declares that enumerated types are rarely needed and uses integer constants to implement them in Eiffel. C++ provides the *enum* construction tool to implement enumerated types. However, the underlying implementation uses integers which makes it possible to perform meaningless operations such as adding two members of the enumeration. Enumerated types are a useful abstraction but are not satisfactorily provided by any of the languages under consideration.

- Subranges

  Subranges are not provided but can be implemented by declaring classes to represent the required range of values. The programmer must supply the code to check that values fall within the desired range.

All three languages provide support for simple types and structured types. True generic types are only supported by Eiffel and a few implementations of C++. Enumerated types and subranges are not supported by any of the languages.

9 Redefinition, redeclaration and the implementation of abstract classes

Before discussing redefinition, redeclaration and the implementation of abstract classes, it is necessary to define clearly the meaning of the terms. Several definitions of the terms are possible. Meyer [3] defines redefinition as changing the implementation, signature (the formal parameters and result type) or specification of an inherited feature. He defines redeclaration as a more general concept including both redefinition and the implementation of a deferred feature inherited from an abstract class. Böszörmenyi [14] uses the following definitions. Redefinition is defined as changing the implementation of a function but keeping the signature and the specification the same. Redeclaration is defined as changing the implementation, signature and specification. Böszörmenyi's definitions are used here because they allow the different aspects of the languages to be discussed more easily. The implementation of an abstract class involves writing code to implement a feature which is named, but not implemented, by the parent class.

- Redefinition—changing the implementation only.

  Oberon-2, Eiffel and C++ all permit redefinition of inherited features. In Oberon-2, any type-bound procedure may be redefined. Eiffel requires that the feature to be redefined is listed in the Redefine section of the class declaration. In C++, only functions declared as virtual in the base class are dynamically bound. These are therefore the only features which can be redefined to ensure that the correct version for any object is chosen at run-time.

  Oberon-2 and C++ allow access to the ancestor's version of the function. In Eiffel, repeated inheritance must be used to obtain two copies. One of the copies is renamed and redefined, the other copy is used to access the ancestor's version.

  Eiffel also permits functions to be redefined as attributes in descendant classes. Attributes cannot be redefined as functions.

- Redeclaration—changing the implementation and the signature.

  Eiffel and C++ permit the redeclaration of functions but Oberon-2 does not. The effects of redeclaration are different in the two languages.

  - In Eiffel, any features being redeclared must be listed in the *Redefine* section of the class declaration. There are strict rules governing changes to the signatures and specification of functions.

    Any change of type in a signature must be to a type which conforms with the original type.

    Changes to the specification of a routine involve changes to the pre- and post-conditions. Pre- and post-conditions are implemented by using the assertion mechanisms provided by the language. The rules governing changes to pre- and post-conditions state that preconditions can be weakened but not strengthened and post conditions may be strengthened not weakened.

    Attributes can also be redeclared in Eiffel. The same rule applies that the new type must conform to the old type.

    The Eiffel rules governing redeclaration are designed to enable the redeclared feature to be used wherever the original form is declared.

  - C++ provides redeclaration by permitting a limited form of function overloading. Overloading means that a single name can be used for several different functions within the same scope. Overloaded functions must differ in the parameters required and may differ in the return type.

    The new function masks the function of the same name declared by the base class. The masked function cannot be accessed directly by objects of the the derived class but can be accessed either by using the scope resolution operator or by assigning the object to an object of its base class. Both of these mechanisms tell the compiler to 'look' in the superclass part of the object to find the required function. Both versions of the function are therefore available so the function name is "overloaded" by redeclaration.

  However, redeclaration does not involve the virtual mechanism so dynamic binding does not occur. The code is chosen statically according to the static type of the object. A redeclared function is effectively a new function so if it is to be redefined in subclasses, it must be declared as virtual.

- The implementation of abstract classes

  All three languages allow the definition of abstract classes where the desired behaviour is specified but not implemented. Each language has its own mechanism for providing this facility. In Oberon-2, abstract behaviour can be specified by declaring empty type-bound procedures. In C++, pure virtual functions are used. In Eiffel the keyword **deferred** is used to replace the body of the feature. The derived class is responsible for supplying the code to implement the behaviour.

  It is impossible in Eiffel and C++ to create instances of deferred classes. Oberon-2 has no such safeguards so it is recommended that the programmer declares a call to the predefined procedure HALT in an empty type-bound procedure.

Eiffel allows the developer to make a wider range of changes to inherited classes. Thereby improving the possibility of code reuse by using inheritance to implement an is-like relationship such as a Queue is-like a List with limited add and remove functions. However, this type of code reuse leads to problems with type conformance which are discussed in section 10.

10 Type checking and type conformance

There are several similarities between the type checking and conformance systems of Oberon-2, Eiffel and C++. These include:

- static type checking,
- conformance is based on the inheritance hierarchy. In simple terms this means that a derived class conforms to its base class.
- assignment compatibility is governed by type conformance.
- type checking is a syntactic mechanism. It is possible to redefine a procedure to perform a totally different function using the same parameter types. There is no semantic checking.

Oberon-2 allows only single inheritance and has strict redefinition rules which ensure that subclasses are subtypes. This makes type checking and type conformance a simple concept for the compiler to enforce. In situations where each subtype must be distinguished, for example when retrieving elements from a heterogeneous list, the programmer must use type guards or type checks to provide dynamic type checking.

Eiffel allows the types of attributes, parameters and return types to be redeclared providing the new parameters conform to the original ones.

Eiffel also allows a derived class to prevent access to features which were exported in its base class. This hiding of features together with the redefinition of the signature of features destroys the subtype relationship between the classes. Eiffel version 3 intended to introduce system validity checks in order to prevent invalid access to features.

C++ has two modes of inheriting, *public* and *private*. The compiler enforces the rule that a class derived by public inheritance cannot prevent access to any of the public fields of its ancestors. This helps to maintain the subtype relationship between the classes. The use of private inheritance allows features to be hidden. There is no longer a subtype relationship between the two classes. The C++ type checking system ensures that the two types no longer conform.

From the above it can be seen that Oberon-2 has adopted a simplistic approach to typing resulting in a greater burden of type checking being placed on the developer. Eiffel allows flexibility for the developer of new classes but has not yet developed a type system to deal with the problems that have arisen. C++ also provides flexibility but has mechanisms to allow the developer to control the type hierarchy. The result is a conceptually simpler static typing checking mechanism in C++ than in Eiffel.

Eiffel also allows some semantic checking. The keyword **ensure** can be used to implement postconditions. The use of postconditions prevents an add feature being redefined to multiply for instance.

11 Accessing the dynamic type of an object

For the purposes of this section, a variable is assumed to be an object implemented as a pointer variable. General assignment rules allow a variable of a subclass to be assigned to a variable of its superclass. The superclass variable then contains more information than is available to the programmer. For instance, a variable of Class customer may be assigned to a variable of class Person. The overdraft ceiling is then inaccessible except via a polymorphic procedure.

A polymorphic procedure redefines a base class routine to provide class dependent behaviour. The redefined routines are then called as required because of the dynamic binding of code. Dynamic binding works well for a routine such as *print* which can be defined in a base class. However, dynamic binding cannot be used to access behaviour known to be present in the actual object but not provided when the base class was implemented.

There are circumstances under which it might be necessary to access subclass fields. For example a system may declare a list of elements of class Person. The list could also be used to store elements of subclasses of Person such as Customer. It might then be necessary to retrieve an instance of class Customer from the list of Person instances. This involves assigning a superclass variable to a subclass variable. Mechanisms to perform this type of assignment are provided by all three languages under discussion.

In order to permit access to the dynamic type of an object, the Eiffel language provides the *reverse assignment attempt.* This is most commonly used when accessing persistent data. The syntax of the call is

```
customer ?= people_list.get(i);
```

where `customer` is of class Customer, `people_list` is a list containing variables conforming to class Person, `get(i)` is the feature which retrieves the i-th element from the list.

If the i-th element was the required class it is assigned to the variable customer. If not, the value of the variable is void.

Eiffel provides two other methods of accessing the dynamic type for use where a large number of types may be found.

1. The library class INTERNAL provides a feature *dynamic_type* which returns an INTEGER. The class INTERNAL is designed to be used as an ancestor class for classes which interface with other languages or database management systems. The use of these features is discouraged because they permit access to the internal representation of an object which breaks the encapsulation.

2. Eiffel classes are descendants of the class ANY. This base class provides generally useful facilities. One of the features is *conforms_to* which makes it possible to ascertain information about the dynamic type of an entity at run-time. However, a call such as `p.conforms_to(c)` returning true does not permit access to any extra features present in c through variable p.

The C++ language does not provide a mechanism to correspond to the Eiffel *reverse assignment attempt.* Similar functionality can be obtained by using pointer casting. This uses the type cast mechanism available in C. Assuming the declarations:

```
person *p;
customer *c;
```

the call `c = (customer *)p` converts a person variable to a customer variable. This mechanism allows unconstrained changes between types which can lead to many problems.

In some circumstances, such as retrieving elements from a heterogeneous list, it might be desirable to apply some constraint to the conversions available by determining the dynamic type of the elements. For example to retrieve objects of type Customer from a list of people. C++ does not provide a procedure to allow the actual type of the object to be ascertained. A programmer must add a tag field to objects which can then be checked to ensure the correct type is chosen.

Oberon-2 provides type tests and type guards to permit access to fields of a subclass variable through a supertype object. Type tests can be used in conjunction with an IF ...THEN ...ELSE statement to provide the equivalent of the Eiffel *reverse assignment attempt.*

From the above, it can be seen that it is possible to access the dynamic type of objects in all three languages. All the mechanisms apply to objects but not to fields of an object. For example, a class Person might declare an address field of class string. A person object could have an instance of a subclass of class string assigned to its address field. This subclass might provide extra string handling facilities. These extra features cannot be accessed through the person variable.

12 System execution

Both Eiffel and C++ require a driver to set the program running. In Eiffel, this is called the root class. The system is started by entering the class name, in lower case, to the unix prompt. This executes the create procedure of the class. Any class can be used as a root class providing it is named as such in the system description document. The C++ driver is a *main* function. This can require parameters which are read from the command line when the program is executed. In both Eiffel and C++ it is necessary to designate a starting point for the system.

The position in Oberon-2 is rather different. The program can be initiated by a call to any parameterless procedure. A system can therefore have many starting points. For example, the Persons module could have contained several test procedures. Any one of these could be called to start the system.

In Eiffel, all procedures are part of a class. In order to access a procedure it is necessary to declare an instance of the class [1]. This instance is a pointer with a void reference. The create function for the class must be invoked before it is possible to access the required procedure using the dot notation. In Oberon-2, procedures can be either bound to a class or a module. Procedures which are

[1]The class IO which contains input and output functions is an exception. Functions from this class can be used in any other class by using the dot notation, For example io.putstring.

bound to a module can be invoked simply by naming the module in the import list and then using the dot notation to identify the procedure.

In C++ functions can be static (available within the source file only), global or be part of a class. Functions which are part of a class in Oberon-2 or C++ are accessed in the same way as Eiffel features.

13 Conclusions

The languages provide all the features required to be considered object-oriented. These features are encapsulation, polymorphism, inheritance and object identity. They also provide the ability to define classes as structured types.

The type compatibility rules of Oberon-2 are simpler than those of C++ and Eiffel because of the rules applied to record extensions. An extended record type is always a subtype of its base record type because programmers cannot change the parameters required by a procedure and cannot change the export status of features. Programmers must use type guards to enforce finer control over type compatibility to ensure that a redefined procedure is not called on a base type object.

In C++ and Oberon-2, it is possible to declare many types in a single module which allows more flexibility to programmers. It is also possible to declare two types in one module but only export one of them for use outside that module. This allows a type to be composed of instances of hidden types. However, the hidden type is not available for reuse which is one of the benefits of object-orientation.

Oberon-2 permits only one interface to a module which means that a type can also only have one public interface. There is no mechanism to allow derived types to access hidden fields of the base type. This restriction can also reduce reusability. Both Eiffel and C++ allow greater access to base class fields to derived classes than to client classes.

Generic types are not a necessary feature of object-oriented languages but they do provide commonly used abstractions. The absence of generic types from Oberon-2 results in the need for repetitive coding.

In all three languages it is possible to retrieve an object of a derived type from an object of its base type. Eiffel and Oberon-2 provide features to do this but C++ programmers must add fields to objects and provide the tests themselves if safe conversions are to be ensured.

Oberon-2 does provide one object-oriented feature which is not available in some other languages. It is possible to have many different starting points in a single system because a main program is not needed.

References

[1] Martin Reiser and Nicholas Wirth. *Programming in Oberon.* Addison-Wesley Publishing Company, New York, 1992.

[2] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, Hemel Hempstead, 1988.

[3] B. Meyer. *Eiffel: The Language.* Prentice Hall, Hemel Hempstead, 1992.

[4] Russel Winder. *Developing C++ Software.* John Wiley and Sons Ltd., Chichester, West Sussex., 1991.

[5] Stanley B. Lippman. *C++ Primer.* Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1991.

[6] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, Reading,Massachusetts, 1990.

[7] Audrey Mayes and Mary Buchanan. The Oberon-2 Language and Environment. Technical Report 190, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1994.

[8] J. A. Mayes and R. Barrett. Eiffel, the universe and everything. TR 138, Hatfield Polytechnic, College Lane, Hatfield, Herts AL10 9AB, 1992.

[9] H. Mossenbock and N. Wirth. Differences between Oberon and Oberon-2. *Structured Programming*, 4 1991.

[10] Setrag Khoshafian and Razmik Abnous. *Object Orientation: Concepts, Languages, Databases, User Interfaces.* Wiley, New York, 1990.

[11] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, Inc, Englewood Cliffs, New Jersey, second edition, 1991.

[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modelling and Design.* Prentice-Hall International Editions, Englewood Cliffs, New Jersey, 1991.

[13] N. Wirth. From Modula to Oberon. *Software - Practice and experience*, 7 1988.

[14] Laszlo Boszormenyi. A Comparison of Modula-3 and Oberon-2. *Structured Programming*, 14, 1993.