Object Libraries

Foundation Classes
User's Guide

Version 6.6.0 / April 2013

NOTICE FOR USERS:

This User Guide is a general instruction for Open CASCADE Technology study. It may be incomplete and even contain occasional mistakes, particularly in examples, samples, etc.

OPEN CASCADE S.A.S. bears no responsibility for such mistakes. If you find any mistakes or imperfections in this document, or if you have suggestions for improving this document, please, contact us and contribute your share to the development of Open CASCADE Technology: bugmaster@opencascade.com

http://www.opencascade.com/contact/

Table of Contents

1. Introduction

1.1. Foundation Classes Overview

This manual explains how to use Open CASCADE Technology (**OCCT**) Foundation Classes. It provides basic documentation on foundation classes. For advanced information on foundation classes and their applications, see our offerings on our web site at www.opencascade.org/support/training/

Foundation Classes provide a variety of general-purpose services such as automated dynamic memory management (manipulation of objects by handle), collections, exception handling, genericity by downcasting and plug-in creation.

Foundation Classes include the following:

Root Classes

They are the basic data types and classes on which all the other classes are built. They provide:

- fundamental types such as Boolean, Character, Integer or Real,
- safe handling of dynamically created objects, ensuring automatic deletion of unreferenced objects (see the Standard_Transient class),
- configurable optimized memory manager increasing the performance of applications that intensively use dynamically created objects,
- extended run-time type information (RTTI) mechanism facilitating the creation of complex programs,
- management of exceptions,
- encapsulation of C++ streams.

Root classes are mainly implemented in the **Standard** and **MMgt** packages.

Strings

Strings are classes that handle dynamically sized sequences of characters based on both ASCII (normal 8-bit character type) and Unicode (16-bit character type).

Strings may also be manipulated by handles, and consequently be shared.

Strings are implemented in the **TCollection** package.

Collections

Collections are the classes that handle dynamically sized aggregates of data.

Collection classes are *generic*, that is, they define a structure and algorithms allowing to hold a variety of objects which do not necessarily inherit from a unique root class (similarly to C++ templates). When you need to use a collection of a given type of object, you must *instantiate* it for this specific type of element. Once this declaration is compiled, all functions available on the generic collection are available on your *instantiated class*.

Collections include a wide range of generic classes such as run-time sized arrays, lists, stacks, queues, sets and hash maps.

Collections are implemented in the **TCollection** and **NCollection** packages.

Collections of Standard Objects

The **TColStd** package provides frequently used instantiations of generic classes from the **TCollection** package with objects from the **Standard** package or strings from the **TCollection** package.

Vectors and Matrices

These classes provide commonly used mathematical algorithms and basic calculations (addition, multiplication, transposition, inversion, etc.) involving vectors and matrices.

Primitive Geometric Types

Open CASCADE Technology primitive geometric types are a STEP-compliant implementation of basic geometric and algebraic entities.

They provide:

- Descriptions of elementary geometric shapes:
    - Points,
    - Vectors,
    - Lines,
    - Circles and conics,
    - Planes and elementary surfaces,
- Positioning of these shapes in space or in a plane by means of an axis or a coordinate system,
- Definition and application of geometric transformations to these shapes:
    - Translations
    - Rotations
    - Symmetries
    - Scaling transformations
    - Composed transformations
- Tools (coordinates and matrices) for algebraic computation.

Common Math Algorithms

Open CASCADE Technology common math algorithms provide a C++ implementation of the most frequently used mathematical algorithms.

These include:

- Algorithms to solve a set of linear algebraic equations,
- Algorithms to find the minimum of a function of one or more independent variables,
- Algorithms to find roots of one, or of a set, of non-linear equations,
- Algorithms to find the eigen-values and eigen-vectors of a square matrix.

Exceptions

A hierarchy of commonly used exception classes is provided, all based on class `Failure`, the root of exceptions.

Exceptions describe exceptional situations, which can arise during the execution of a function. With the raising of an exception, the normal course of program execution is abandoned. The execution of actions in response to this situation is called the treatment of the exception.

Quantities

These are various classes supporting date and time information and fundamental types representing most physical quantities such as length, area, volume, mass, density, weight, temperature, pressure etc.

Application services

Foundation Classes also include implementation of several low-level services that facilitate the creation of customizable and user-friendly applications with Open CASCADE Technology. These include:

- Unit conversion tools, providing a uniform mechanism for dealing with quantities and associated physical units: check unit compatibility, perform conversions of values between different units and so on (see package UnitsAPI).
- Basic interpreter of expressions that facilitates the creation of customized scripting tools, generic definition of expressions and so on (see package ExprIntrp)
- Tools for dealing with configuration resource files (see package Resource) and customizable message files (see package Message), making it easy to provide a multi-language support in applications
- Progress indication and user break interfaces, giving a possibility even for low-level algorithms to communicate with the user in a universal and convenient way.

For a detailed description of all the Foundation Classes, see the *Foundation Classes Reference Manual.*

1.2. Fundamental Concepts

An object-oriented language structures a system around data types rather than around the actions carried out on this data. In this context, an **object** is an **instance** of a data type and its definition determines how it can be used. Each data type is implemented by one or more classes, which make up the basic elements of the system.

In Open CASCADE Technology the classes are usually defined using CDL (CASCADE Definition Language) that provides a certain level of abstraction from pure C++ constructs and ensures a definite level of similarity in the implementation of classes. See *CDL User's Guide* for more details.

This chapter introduces some basic concepts most of which are directly supported by CDL and used not only in Foundation Classes, but throughout the whole OCCT library.

1.2.1. Modules and toolkits

The whole OCCT library is organized in a set of modules. The first module, providing most basic services and used by all other modules, is called Foundation Classes and described by this manual.

Every module consists primarily of one or several toolkits (though it can also contain executables, resource units etc.). Physically a toolkit is represented by a shared library (e.g. .so or .dll). The toolkit is built from one or several packages.

1.2.2. Packages

A **package** groups together a number of classes which have semantic links. For example, a geometry package would contain Point, Line, and Circle classes. A package can also contain enumerations, exceptions and package methods (functions). In practice, a class name is prefixed with the name of its package e.g.

Geom_Circle.

Data types described in a package *may* include one or more of the following data types:

- Enumerations
- Object classes
- Exceptions
- Pointers to other object classes

Inside a package, two data types *cannot* bear the same name.

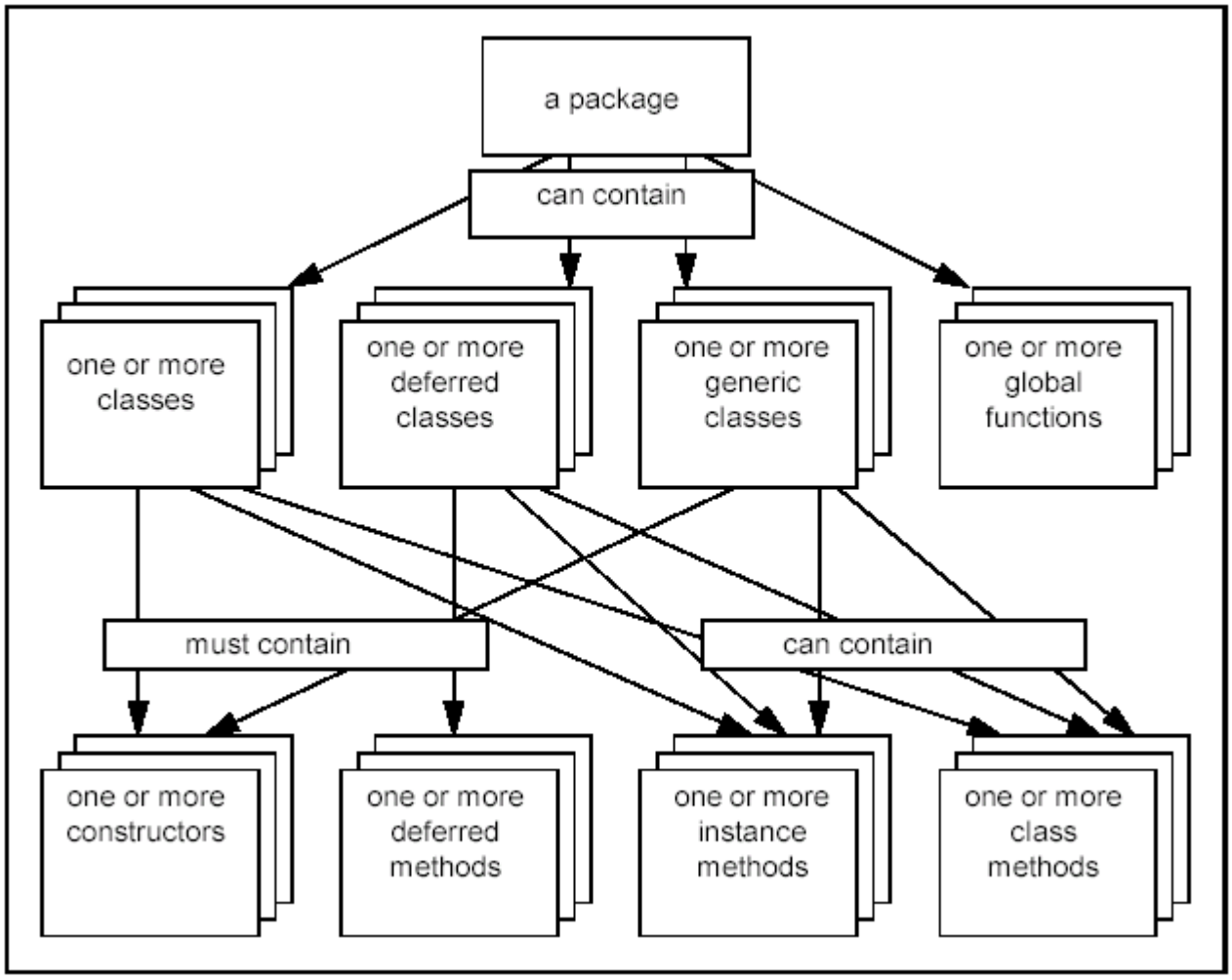

Figure 2. Contents of a package

Methods are either **functions** or **procedures**. Functions return an object, whereas procedures only communicate by passing arguments. In both cases, when the transmitted object is an instance manipulated by a handle, its identifier is passed. There are three categories of methods:

Object constructor	Creates an instance of the described class. A class will have one or more object constructors with various different arguments or none.
Instance method	Operates on the instance which owns it.
Class method	Does not work on individual instances, only on the class itself.

1.2.3. Classes

The fundamental software component in object-oriented software development is the class. A class is the implementation of a **data type**. It defines its **behavior** (the services offered by its functions) and its **representation** (the data structure of the class – the fields which store its data).

Categories of Classes

Classes fall into three categories:

- Ordinary classes
- Deferred classes
- Generic classes

A **deferred class** cannot be instantiated. The purpose of having such classes is to have a given behavior shared by a hierarchy of classes and dependent on the implementation of the descendents. This is a way of guaranteeing a certain base of inherited behavior common to all the classes based on a particular deferred class. The C++ equivalent of a deferred CDL class is an abstract class.

A **generic class** offers a set of functional behaviors to manipulate other data types. Instantiation of a generic class requires that a data type is given for its argument(s). The generic classes in CDL perform the same mission as template classes in C++.

1.2.4. Genericity

Generic classes are implemented in two steps. First you declare the generic class to establish the model, then you instantiate this class by giving information about the generic types.

Declaring a Generic Class

The generic classes in Open CASCADE Technology are similar by their intent to C++ templates with explicit instantiation.

A generic class is declared in CDL as operating on data items of non-fixed types which are declared as arguments of the generic class. It is possible to put a restriction on these data types to be of subtype of some definite class. Definition of the generic class does not create new class type in C++ terms; it only defines a pattern for generation (instantiation) of the real classes.

Instantiation of a Generic Class

When a generic class is instantiated, its argument types are substituted by actually existing data types (elementary types or classes). The result of instantiation is a new C++ class with an arbitrary name (specified in the instantiating declaration). By convention, the name of the instantiated class is usually constructed from the name of the generic class and names of actual argument types. As for any other class, the name of the class instantiating a generic type is prefixed by the name of the package in which instantiation is declared.

Example:

```
class Array1OfReal instantiates Array1 from TCollection (Real);
```

This declaration located in a CDL file of the TColStd package defines a new C++ class TColStd_Array1OfReal as the instantiation of generic class TCollection_Array1 for Real values.

More than one class can be instantiated from the same generic class with the same argument types. Such classes will be identical by implementation, but considered as two different classes by C++.

No class can inherit from a generic class.

A generic class can be a deferred class. A generic class can also accept a deferred class as its argument. In both these cases, any class instantiated from it will also be deferred. The resulting class can then be inherited by another class.

Nested Generic Classes

It often happens that many classes are linked by a common generic type. This is the case when a base structure furnishes an iterator. In this context, it is necessary to make sure that the group of linked generic classes is indeed instantiated for the same type of object. In order to group the instantiation, you may declare certain classes as being nested.

When generic class is instantiated, its nested classes are instantiated as well. The name of the instantiation of the nested class is constructed from the name of that nested class and name of the main generic class, connected by 'Of'.

Example:

```
class MapOfReal instantiates Map from TCollection (Real,MapRealHasher);
```

This declaration in TColStd defines not only class TColStd_MapOfReal, but also class TColStd_MapIteratorOfMapOfReal which is instantiated from nested class MapIterator of the generic class TCollection_Map. Note that instantiation of the nested class is separate class, it is not nested class to the instantiation of the main class.

Nested classes, even though they are described as non-generic classes, are generic by construction being inside the class they are a member of.

1.2.5. Inheritance

The purpose of inheritance is to reduce the development workload. The inheritance mechanism allows a new class to be declared already containing the characteristics of an existing class. This new class can then be rapidly specialized for the task in hand. This avoids the necessity of developing each component "from scratch".

For example, having already developed a class BankAccount you could quickly specialize new classes - SavingsAccount, LongTermDepositAccount, MoneyMarketAccount, RevolvingCreditAccount, etc....

The corollary of this is that when two or more classes inherit from a parent (or ancestor) class, all these classes guarantee as a minimum the behavior of their parent (or ancestor). For example, if the parent class BankAccount contains the method Print which tells it to print itself out, then all its descendent classes guarantee to offer the same service.

One way of ensuring the use of inheritance is to declare classes at the top of a hierarchy as being **deferred**. In such classes, the methods are not implemented. This forces the user to create a new class which redefines the methods. This is a way of guaranteeing a certain minimum of behavior among descendent classes.

1.2.6. Categories of Data Types

The data types in Open CASCADE Technology fall into two categories:

- Data types manipulated by handle (or reference)
- Data types manipulated by value

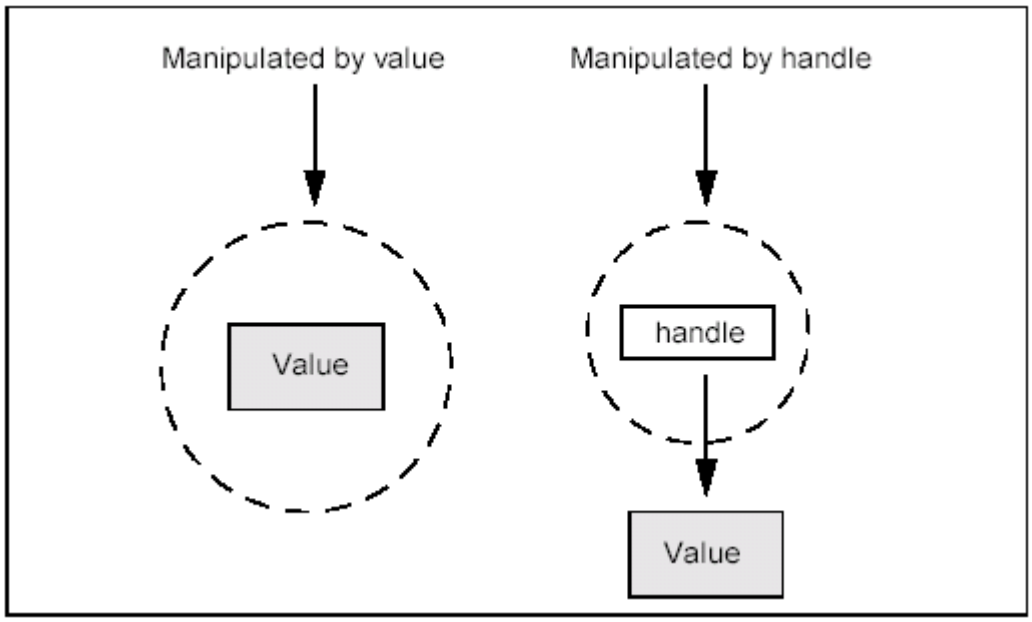

Figure 1. Manipulation of data types

A data type is implemented as a class. The class not only defines its data representation and the methods available on instances, but it also suggests how the instance will be manipulated.

- A variable of a type manipulated by value contains the instance itself.
- A variable of a type manipulated by handle contains a reference to the instance.

The first examples of types manipulated by values are the predefined **primitive types**: Boolean, Character, Integer, Real etc.

A variable of a type manipulated by handle which is not attached to an object is said to be **null**. To reference an object, we instantiate the class with one of its constructors. For example, in C++:

Example

```
Handle(myClass) m = new myClass;
```

In Open CASCADE Technology, the Handles are specific classes that are used to safely manipulate objects allocated in the dynamic memory by reference, providing reference counting mechanism and automatic destruction of the object when it is not referenced.

1.2.7. Exceptions

The behavior of any object is implemented by the methods, which were defined in its class declaration. The definition of these methods includes not only their signature (their programming interface) but also their domain of validity.

This domain is expressed by **exceptions**. Exceptions are raised under various error conditions. This mechanism is a safeguard of software quality.

1.2.8. Persistence and Data Schema

The data schema is the structure used by an application to store its data. Data schemas consist of persistent classes.

An object is called **persistent** if it can be permanently stored. Thus, the object can be reused at a later date by the application, which created it, or by another application.

In order for an object to be persistent for CDL, its type must be declared as inheriting from the class Standard_Persistent or have a parent class inheriting from the Standard_Persistent class. Note that classes inheriting from Standard_Persistent are handled by a reference.

Objects instantiated from classes which inherit from the Standard_Storable class cannot themselves be stored individually, but they can be stored as fields of an object which inherits from Standard_Persistent. Note that objects inheriting from Standard_Storable are handled by a value.

2. Basics

This chapter deals with basic services such as memory management, programming with handles, primitive types, exception handling, genericity by downcasting and plug-in creation.

2.1. Data Types

2.1.1. Primitive Types

The primitive types are predefined in the language and they are **manipulated by value**.

Some of these primitives inherit from the **Storable** class. This means they can be used in the implementation of persistent objects, either contained in entities declared within the methods of the object, or they form part of the internal representation of the object.

The primitives inheriting from Standard_Storable are the following:

Boolean Is used to represent logical data. It may have only two values: Standard_True and Standard_False.

Character Designates any ASCII character.

ExtCharacter Is an extended character.

Integer Is a whole number.

Real Denotes a real number (i.e. one with whole and a fractional part, either of which may be null).

ShortReal Real with a smaller choice of values and memory size.

There are also non-Storable primitives. They are:

CString Is used for literal constants.

ExtString Is an extended string.

Address Represents a byte address of undetermined size.

The services offered by each of these types are described in the **Standard** Package.

The table below presents the equivalencies existing between C++ fundamental types and OCCT primitive types.

Table 1: Equivalence between C++ Types and OCCT Primitive Types

C++ Types	OCCT Types
int	Standard_Integer
double	Standard_Real
float	Standard_ShortReal
unsigned int	Standard_Boolean

	Standard_False = 0 Standard_True = 1
char	Standard_Character
short	Standard_ExtCharacter
char*	Standard_CString
void*	Standard_Address
short*	Standard_ExtString

* pointer

Description reminder of the classes listed above:

Standard_Integer:

fundamental type representing 32-bit integers yielding negative, positive or null values. **Integer** is implemented as a **typedef** of the C++ **int** fundamental type. As such, the algebraic operations +, -, *, / as well as the ordering and equivalence relations <, <=, ==, !=, >=, > are defined on it.

Standard_Real:

fundamental type representing real numbers with finite precision and finite size. **Real** is implemented as a **typedef** of the C++ **double** (double precision) fundamental type. As such, the algebraic operations +, -, *, /, unary- and the ordering and equivalence relations <, <=, ==, !=, >=, > are defined on reals.

Standard_ShortReal:

fundamental type representing real numbers with finite precision and finite size. **ShortReal** is implemented as a **typedef** of the C++ **float** (simple precision) fundamental type. As such, the algebraic operations +, -, *, /, unary- and the ordering and equivalence relations <, <=, ==, !=, >=, > are defined on reals.

Standard_Boolean:

Boolean is a fundamental type representing logical expressions. It has two values, false and true. **Boolean** is implemented as a **typedef** of the C++ **unsigned int** fundamental type. As such, the algebraic operations and, or, xor, not as well as equivalence relations ==, != are defined on Booleans.

Standard_Character:

Character is a fundamental type representing the normalized ASCII character set. It may be assigned the values of the 128 ASCII characters. **Character** is implemented as a **typedef** of the C++ **char** fundamental type. As such, the ordering and equivalence relations <, <=, ==, !=, >=, > are defined on characters using the order of the ASCII chart (ex: A< B).

Standard_ExtCharacter:

ExtCharacter is a fundamental type representing the Unicode character set. It is a 16-bit character type. **ExtCharacter** is implemented as a **typedef** of the C++ **short** fundamental type. As such, the ordering and equivalence relations <, <=, ==, !=, >=, > are defined on extended characters using the order of the UNICODE chart (ex: A< B).

Standard_CString:

CString is a fundamental type representing string literals. A string literal is a sequence of ASCII (8 bits) characters enclosed in double quotes. **CString** is implemented as a **typedef** of the C++ **char*** fundamental type.

Standard_Address :

Address is a fundamental type representing a generic pointer. **Address** is implemented as a **typedef** of the C++ **void*** fundamental type.

Standard_ExtString :

ExtString is a fundamental type representing string literals as sequences of Unicode (16 bits) characters. **ExtString** is implemented as a **typedef** of the C++ **short*** fundamental type.

2.1.2. Types manipulated by value

There are three categories of types which are manipulated by value:

- Primitive types
- Enumerated types
- Types defined by classes not inheriting from Standard_Persistent or Standard_Transient, whether directly or not

Types which are manipulated by value behave in a more direct fashion than those manipulated by handle and thus can be expected to perform operations faster, but they cannot be stored independently in a file.

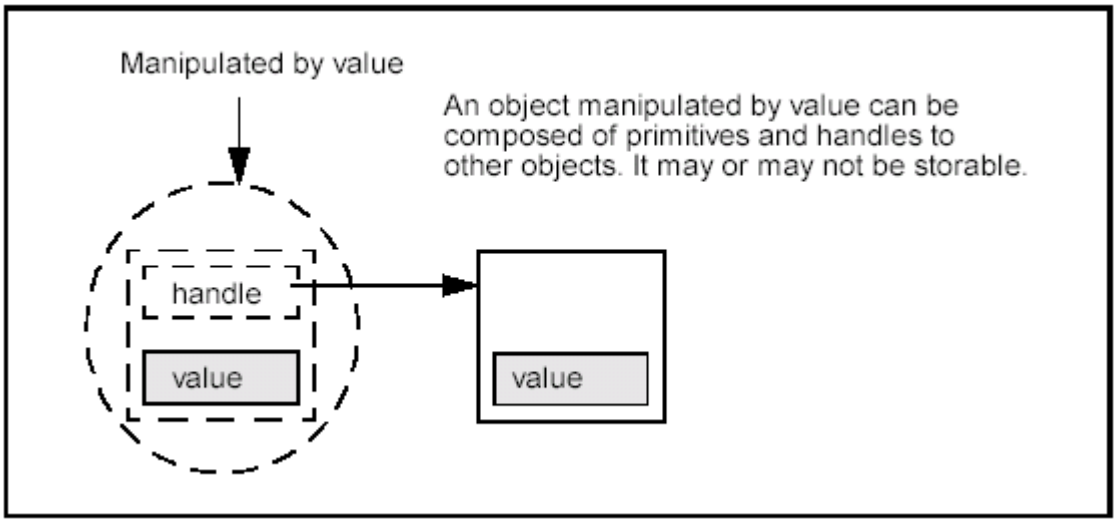

Figure 4. Manipulation of a data type by value

Types that are known to the schema (i.e. they are either **primitives** or they inherit from **Storable**) and are manipulated by value, can be stored inside a persistent object as part of the representation. Only in this way can a “manipulated by value” object be stored in a file.

2.1.3. Types manipulated by reference (handle)

There are two categories of types which are manipulated by handle:

- Types defined by classes inheriting from the **Persistent** class, which are therefore storable in a file.
- Types defined by classes inheriting from the **Transient** class.

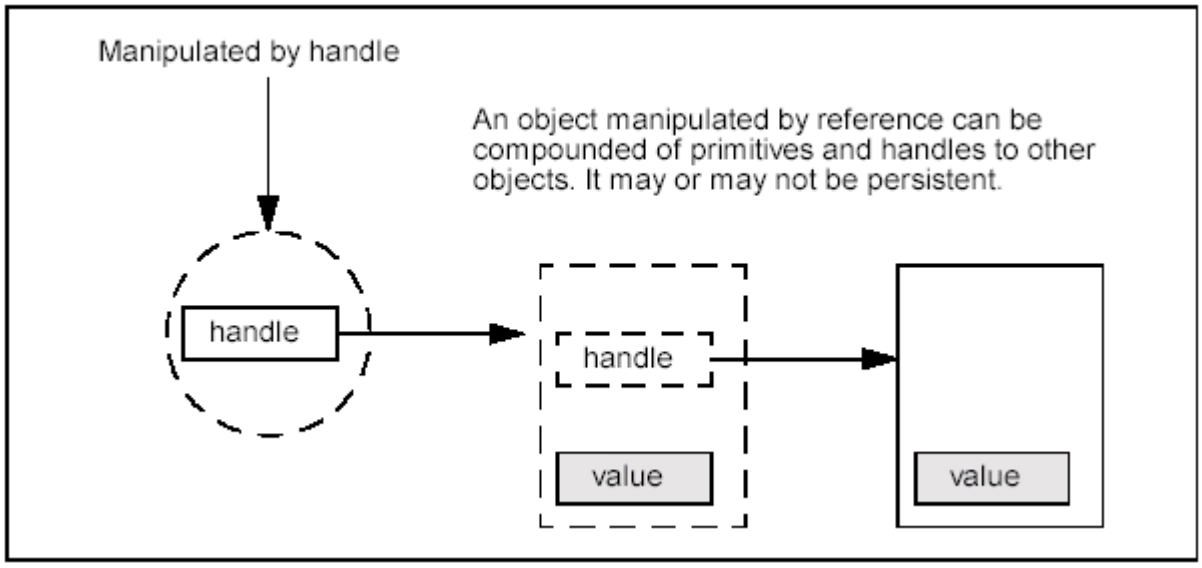

Figure 5. Manipulation of a data type by reference

2.1.4. Summary of properties

	Manipulated by handle	Manipulated by value
storable	***Persistent***	**Primitive**, **Storable** (storable if nested in a persistent class)
temporary	**Transient**	**Other**

Figure 6. Summary of the relationship for the various data types between how they are handled and their storability.

2.2. Programming with Handles

2.2.1. Handle Definition

A handle may be compared with a C++ pointer. Several handles can reference the same object. Also, a single handle may reference several objects, but only one at a time. To have access to the object it refers to, the handle must be de-referenced just as with a C++ pointer.

Transient and Persistent classes may be manipulated either with handles or with values. Handles which reference non-persistent objects are called non-storable handles; therefore, a persistent object cannot contain a non-storable handle.

Organization of Classes

Classes used with handles are persistent or transient.

Classes that inherit from Standard_Transient are transient while classes that inherit from Standard_Persistent are persistent.

In this chapter we will discuss only transient classes and relevant handles. Persistent classes and their handles are organized in a similar manner.

Class Standard_Transient is a root of a big hierarchy of OCCT classes that are said to be operable by handles. It provides a reference counter field, inherited by all its descendant classes, that is used by associated Handle() classes to track a number of handles pointing to this instance of the object.

For every class derived (directly or indirectly) from Transient, CDL extractor creates associated class Handle() whose name is the same as the name of that class prefixed by “Handle_”. Open CASCADE Technology provides pre-processor macro Handle() that produces a name of a Handle() class for a given transient class name.

Using a Handle

A handle is characterized by the object it references.

Before performing any operation on a transient object, you must declare the handle.

For example, if Point and Line are two transient classes from the Geom package, you would write:

Example

```
Handle(Geom_Point) p1, p2;
Handle(Geom_Line) aLine;
```

Declaring a handle creates a null handle that does not refer to any object. The handle may be checked to be null by its method IsNull().To nullify a handle, use method Nullify().

To initialize a handle, either a new object should be created or the value of another handle can be assigned to it, on condition that their types are compatible.

NOTE

Handles should only be used for object sharing. For all local operations, it is advisable to use classes manipulated by values.

2.2.2. Type Management

General

Open CASCADE Technology provides a means to describe the hierarchy of data types in a generic way, with a possibility to check the exact type of the given object at run-time (similarly to C++ RTTI). For every class type derived from Standard_Transient, CDL extractor creates a code instantiating single instance of the class Standard_Type (type descriptor) that holds information on that type: its name and list of ancestor types.

That instance (actually, a handle on it) is returned by the virtual method DynamicType() of the class derived from Standard_Transient. The other virtual method IsKind() provides a means to check whether a given object has specified type or inherits it.

In order to refer to the type descriptor object for a given class type, use macros STANDARD_TYPE() with argument being a name of the class.

Type Conformity

The type used in the declaration of a handle is the static type of the object, the type seen by the compiler. A handle can reference an object instantiated from a subclass of its static type. Thus, the dynamic type of an object (also called the actual type of an object) can be a descendant of the type which appears in the handle declaration through which it is manipulated.

Consider the persistent class *CartesianPoint*, a sub-class of *Point*; the rule of type conformity can be illustrated as follows:

Example

```
Handle (Geom_Point) p1;
Handle (Geom_CartesianPoint) p2;
p2 = new Geom_CartesianPoint;
p1 = p2;  // OK, the types are compatible
```

The compiler sees p1 as a handle to *Point* though the actual object referenced by *p1* is of the *CartesianPoint* type.

Explicit Type Conversion

According to the rule of type conformity, it is always possible to go up the class hierarchy through successive assignments of handles. On the other hand, assignment does not authorize you to go down the hierarchy. Consequently, an explicit type conversion of handles is required.

A handle can be converted explicitly into one of its sub-types if the actual type of the referenced object is a descendant of the object used to cast the handle. If this is not the case, the handle is nullified (explicit type conversion is sometimes called a “safe cast”). Consider the example below.

Example

```
Handle (Geom_Point) p1;
Handle (Geom_CartesianPoint) p2, p3;
p2 = new Geom_CartesianPoint;
p1 = p2; // OK, standard assignment
p3 = Handle (Geom_CartesianPoint)::DownCast (p1);
// OK, the actual type of p1 is CartesianPoint, although
the static type of the handle is Point
```

If conversion is not compatible with the actual type of the referenced object, the handle which was “cast” becomes null (and no exception is raised). So, if you require reliable services defined in a sub-class of the type seen by the handle (static type), write as follows:

Example

```
void MyFunction (const Handle(A) & a)
{
  Handle(B) b = Handle(B)::Downcast(a);
  if (! b.IsNull()) {
    // we can use “b” if class B inherits from A
  }
  else {
    // the types are incompatible
  }
}
```

Downcasting is used particularly with collections of objects of different types; however, these objects should inherit from the same root class.

For example, with a sequence of SequenceOfTransient transient objects and two classes A and B that both inherit from Standard_Transient, you get the following syntax:

Example

```
Handle (A) a;
```

```
Handle (B) b;
Handle (Standard_Transient) t;
SequenceOfTransient s;
a = new A;
s.Append (a);
b = new B;
s.Append (b);
t = s.Value (1);
// here, you cannot write:
// a = t; // ERROR !
// so you downcast:
a = Handle (A)::Downcast (t)
if (! a.IsNull()) {
        // types are compatible, you can use a
}
else {
       // the types are incompatible
}
```

2.2.3. Using Handles to Create Objects

To create an object which is manipulated by handle, declare the handle and initialize it with the standard C++ **new** operator, immediately followed by a call to the constructor. The constructor can be any of those specified in the source of the class from which the object is instanced.

Example

```
Handle (Geom_CartesianPoint) p;
p = new Geom_CartesianPoint (0, 0, 0);
```

Unlike for a pointer, the **delete** operator does not work on a handle; the referenced object is automatically destroyed when no longer in use.

2.2.4. Invoking Methods

Once you have a handle on a persistent or transient object, you can use it like a pointer in C++. To invoke a method which acts on the referenced object, you translate this method by the standard arrow **->** operator, or alternatively, by function call syntax when this is available.

To test or to modify the state of the handle, the method is translated by the dot "." operator.

The example below illustrates how to access the coordinates of an (optionally initialized) point object:

Example

```
Handle (Geom_CartesianPoint) centre;
Standard_Real x, y, z;
if (centre.IsNull()) {
  centre = new PGeom_CartesianPoint (0, 0, 0);
}
centre->Coord(x, y, z);
```

The example below illustrates how to access the type object of a Cartesian point:

Example

```
Handle(Standard_Transient) p = new Geom_CartesianPoint(0.,0.,0.);
if ( p->DynamicType() == STANDARD_TYPE(Geom_CartesianPoint) )
  cout << "Type check OK" << endl;
else
  cout << "Type check FAILED" << endl;
```

NullObject exception will be raised if a field or a method of an object is accessed via a *Null* handle.

Invoking Class Methods

A class method is called like a static C++ function, i.e. it is called by the name of the class of which it is a member, followed by the "`::`" operator and the name of the method.

Example: finding the maximum degree of a Bezier curve:

```
Standard_Integer n;
n = Geom_BezierCurve::MaxDegree();
```

2.2.5. Handle de-allocation

Before you delete an object, you must ensure it is no longer referenced. To reduce the programming load related to this management of object life, the delete function in Open CASCADE Technology is secured by a **reference counter** of classes manipulated by handle**.** A handle automates the reference counter management and automatically deletes an object when it is no longer referenced. Normally you never call the delete operator explicitly on instances of subclasses of `Standard_Transient`.

When a new handle to the same object is created, the reference counter is incremented. When the handle is destroyed, nullified, or reassigned to another object, that counter is decremented. The object is automatically deleted by the handle when reference counter becomes 0.

The principle of allocation can be seen in the example below.

Example

```
...
{
Handle (TColStd_HSequenceOfInteger) H1 = new TColStd_HSequenceOfInteger;

  // H1 has one reference and corresponds to 48 bytes of memory
  {
    Handle (TColStd_HSequenceOfInteger) H2;
    H2 = H1; // H1 has two references
    if (argc == 3) {
      Handle (TColStd_HSequenceOfInteger) H3;
      H3 = H1;
      // Here, H1 has three references
      ...
    }
    // Here, H1 has two references
  }
  // Here, H1 has 1 reference
}
// Here, H1 has no reference and the referred
// TColStd_HSequenceOfInteger object is deleted
```

Cycles

Cycles appear if two or more objects reference each other by handles (stored as fields). In this condition automatic destruction will not work.

Consider for example a graph, whose objects (primitives) have to know the graph object to which they belong, i.e. a primitive must have a reference to complete graph object. If both primitives and the graph are manipulated by handle and they refer to each other by keeping a handle as a field, the cycle appears.

The graph object will not be deleted when the last handle to it is destructed in the application, since there are handles to it stored inside its own data structure (primitives).

There are two approaches how to avoid such situation:

- Use C++ pointer for one kind of references, e.g. from a primitive to the graph
- Nullify one set of handles (e.g. handles to a graph in primitives) when a graph object needs to be destroyed

2.2.6. Creating Transient Classes without CDL

Though generation of Handle class and related C++ code is normally performed by CDL extractor, it is also possible to define a class managed by handle without CDL. To facilitate that, several macros are provided in the file Standard_DefineHandle.hxx:

DEFINE_STANDARD_HANDLE(class_name,ancestor_name)

This macro declares Handle class for a class *class_name* that inherits class *ancestor_name* (for instance, Standard_Transient). This macro should be put in a header file; the declaration of the handle to a base class must be available (usually put before or after the declaration of the class *class_name*, or into a separate header file).

IMPLEMENT_STANDARD_HANDLE(class_name,ancestor_name)

This macro implements method DownCast() of the Handle class. Should be located in a C++ file (normally the file where methods of the class *class_name* are implemented).

DEFINE_STANDARD_RTTI(class_name)

Declares methods required for RTTI in the class *class_name* declaration; should be in public: section.

IMPLEMENT_STANDARD_RTTIEXT(class_name,ancestor_name)

Implements above methods. Usually put into the C++ file implementing class class_name.

Note that it is important to ensure correctness of macro arguments, especially the ancestor name, otherwise the definition may be inconsistent (no compiler warnings will be issued in case of mistake).

Example:

Appli_ExtSurface.hxx file:

```
#include <Geom_Surface.hxx>
class Appli_ExtSurface : public Geom_Surface
{
. . .
public:
  DEFINE_STANDARD_RTTI(Appli_ExtSurface)
}
DEFINE_STANDARD_HANDLE(Appli_ExtSurface,Geom_Surface)
```

Appli_ExtSurface.cxx file:

```
#include <Appli_ExtSurface.hxx>
IMPLEMENT_STANDARD_HANDLE(Appli_ExtSurface,Geom_Surface)
IMPLEMENT_STANDARD_RTTIEXT(Appli_ExtSurface,Geom_Surface)
```

2.3. Memory Management in Open CASCADE Technology

In the course of a work session, geometric modeling applications create and delete a considerable number of C++ objects allocated in the dynamic memory (heap). In this context, performance of standard functions for allocating and de-allocating memory may be not sufficient. For this reason, Open CASCADE Technology employs a specialized memory manager implemented in the Standard package.

2.3.1. Usage

To use the Open CASCADE Technology memory manager to allocate memory in a C code, just use method Standard::Allocate() instead of malloc() and method Standard::Free() instead of free(). In addition, method Standard::Reallocate() is provided to replace C function realloc().

In C++, operators new() and delete() for a class may be defined so as to allocate memory using Standard::Allocate() and free it using Standard::Free(). In that case all objects of that class and all inherited classes will be allocated using the OCCT memory manager.

CDL extractor defines new() and delete() in this way for all classes declared with CDL. Thus all OCCT classes (apart from a few exceptions) are allocated using the OCCT memory manager.

Since operators new() and delete() are inherited, this is also true for any class derived from an OCCT class, for instance, for all classes derived from Standard_Transient.

NOTE

It is possible (though not recommended unless really unavoidable) to redefine new() and delete() functions for some class inheriting Standard_Transient. If that is done, the method Delete() should be also redefined to apply operator *delete* to *this* pointer. This will ensure that appropriate delete() function will be called, even if the object is manipulated by a handle to a base class.

2.3.2. Configuring the memory manager

The OCCT memory manager may be configured to apply different optimization techniques to different memory blocks (depending on their size), or even to avoid any optimization and use C functions malloc() and free() directly.

The configuration is defined by numeric values of the following environment variables:

- MMGT_OPT: if set to 0 (default) every memory block is allocated in C memory heap directly (via malloc() and free() functions). In this case, all other options except MMGT_CLEAR are ignored; if set to 1 the memory manager performs optimizations as described below; if set to 2, Intel ® TBB optimized memory manager is used.
- MMGT_CLEAR: if set to 1 (default), every allocated memory block is cleared by zeros; if set to 0, memory block is returned as it is.

- MMGT_CELLSIZE: defines the maximal size of blocks allocated in large pools of memory. Default is 200.
- MMGT_NBPAGES: defines the size of memory chunks allocated for small blocks in pages (operating-system dependent). Default is 1000.
- MMGT_THRESHOLD: defines the maximal size of blocks that are recycled internally instead of being returned to the heap. Default is 40000.
- MMGT_MMAP: when set to 1 (default), large memory blocks are allocated using memory mapping functions of the operating system; if set to 0, they will be allocated in the C heap by malloc().
- MMGT_REENTRANT: when set to 1 (default), all calls to the optimized memory manager will be secured against possible simultaneous access from different execution threads. This variable should be set in any multithreaded application that uses an optimized memory manager (MMGT_OPT=1) and has more than one thread potentially calling OCCT functions. If set to 0, OCCT memory management and exception handling routines will skip the code protecting from possible concurrency in multi-threaded environment. This can yield some performance gain in some applications, but can lead to unpredictable results if used in a multithreaded application.

  NOTE

  For applications that use OCCT memory manager from more than one thread, on multiprocessor hardware, it is recommended to use options MMGT_OPT=2 and MMGT_REENTRANT=1.).

2.3.3. Implementation details

When MMGT_OPT is set to 1, the following optimization techniques are used:

- Small blocks with a size less than MMGT_CELLSIZE, are not allocated separately. Instead, a large pools of memory are allocated (the size of each pool is MMGT_NBPAGES pages). Every new memory block is arranged in a spare place of the current pool. When the current memory pool is completely occupied, the next one is allocated, and so on.

  In the current version memory pools are never returned to the system (until the process finishes). However, memory blocks that are released by the method Standard::Free() are remembered in the free lists and later reused when the next block of the same size is allocated (recycling).
- Medium-sized blocks, with a size greater than MMGT_CELLSIZE but less than MMGT_THRESHOLD, are allocated directly in the C heap (using malloc() and free()). When such blocks are released by the method Standard::Free() they are recycled just like small blocks.

  However, unlike small blocks, the recycled medium blocks contained in the free lists (i.e. released by the program but held by the memory manager) can be returned to the heap by method Standard::Purge().
- Large blocks with a size greater than MMGT_THRESHOLD, including memory pools used for small blocks, are allocated depending on the value of MMGT_MMAP: if it is 0, these blocks are allocated in the C heap; otherwise they are allocated using operating-system specific functions managing memory mapped files.

Large blocks are returned to the system immediately when Standard::Free() is called.

Benefits and drawbacks

The major benefit of the OCCT memory manager is explained by its recycling of small and medium blocks that makes an application work much faster when it constantly allocates and frees multiple memory blocks of similar sizes. In practical situations, the real gain on the application performance may be up to 50%.

The associated drawback is that recycled memory is not returned to the operating system during program execution. This may lead to considerable memory consumption and even be misinterpreted as a memory leak. To minimize this effect, the method Standard::Purge() shall be called after the completion of memory-intensive operations.

The overhead expenses induced by the OCCT memory manager are:

- size of every allocated memory block is rounded up to 8 bytes (when MMGT_OPT is 0 (default), the rounding is defined by the CRT; the typical value for 32-bit platforms is 4 bytes)
- additional 4 bytes (or 8 on 64-bit platforms) are allocated in the beginning of every memory block to hold its size (or address of the next free memory block when recycled in free list) only when MMGT_OPT is 1

Note that these overheads may be greater or less than overheads induced by the C heap memory manager, so overall memory consumption may be greater in either optimized or standard modes, depending on circumstances.

As a general rule, it is advisable to allocate memory through significant blocks. In this way, you can work with blocks of contiguous data, and processing is facilitated for the memory page manager.

In multithreaded mode (MMGT_REENTRANT=1), the OCCT memory manager uses mutex to lock access to free lists, therefore it may have less performance than non-optimized mode in situations when different threads often make simultaneous calls to the memory manager. The reason is that modern implementations of malloc() and free() employ several allocation arenas and thus avoid delays waiting mutex release, which are possible in such situations.

2.4. Exception Handling

Exception handling provides a means of transferring control from a given point in a program being executed to an **exception handler** associated with another point previously executed.

A method may raise an exception which interrupts its normal execution and transfers control to the handler catching this exception.

Open CASCADE Technology provides a hierarchy of exception classes with a root class being class Standard_Failure from the Standard package. The CDL extractor generates exception classes with standardized interface.

Open CASCADE Technology also provides support for converting system signals (such as access violation or division by zero) to exceptions, so that such situations can be safely handled with the same uniform approach.

However, in order to support this functionality on various platforms, some special methods and workarounds are used. Though the implementation details are hidden and handling of OCCT exceptions is done basically in the same way as with C++, some peculiarities of this approach shall be taken into account and some rules must be respected.

The following paragraphs describe recommended approaches for using exceptions when working with Open CASCADE Technology.

2.4.1. Raising an Exception

"C++ like" Syntax

To raise an exception of a definite type method Raise() of the appropriate exception class shall be used.

Example

```
DomainError::Raise("Cannot cope with this condition");
```

raises an exception of DomainError type with the associated message "Cannot cope with this condition", the message being optional. This exception may be caught by a handler of some DomainError type as follows:

Example

```
try {
  OCC_CATCH_SIGNALS
  // try block
}
catch(DomainError) {
// handle DomainError exceptions here
}
```

Regular usage

Exceptions should not be used as a programming technique, to replace a "goto" statement for example, but as a way to protect methods against misuse. The caller must make sure its condition is such that the method can cope with it.

Thus,

- No exception should be raised during normal execution of an application.
- A method which may raise an exception should be protected by other methods allowing the caller to check on the validity of the call.

For example, if you consider the TCollection_Array1 class used with:

- a Value function to extract an element

- a Lower function to extract the lower bound of the array
- an Upper function to extract the upper bound of the array,

then, the Value function may be implemented as follows:

Example

```
Item TCollection_Array1::Value (const Standard_Integer&index) const
{
  // where r1 and r2 are the lower and upper bounds of the array
  if(index < r1 || index > r2) {
    OutOfRange::Raise("Index out of range in Array1::Value");
  }
  return contents[index];
}
```

Here validity of the index is first verified using the Lower and Upper functions in order to protect the call.

Normally the caller ensures the index being in the valid range before calling Value(). In this case the above implementation of Value is not optimal since the test done in Value is time-consuming and redundant.

It is a widely used practice to include that kind of protections in a debug build of the program and exclude in release (optimized) build. To support this practice, the macros Raise_if() are provided for every OCCT exception class:

<ErrorTypeName>_Raise_if(condition, "Error message");

where ErrorTypeName is the exception type, condition is the logical expression leading to the raise of the exception, and Error message is the associated message.

The entire call may be removed by defining one of the pre-processor symbols No_Exception or No_<ErrorTypeName> at compile-time:

Example

```
#define No_Exception /* remove all raises */
```

Using this syntax, the Value function becomes:

Example

```
Item TCollection_Array1::Value (const Standard_Integer&index) const
```

```
{
  OutOfRange_Raise_if(index < r1 || index > r2,
                      “index out of range in Array1::Value”);
  return contents[index];
}
```

2.4.2. Handling an Exception

When an exception is raised, control is transferred to the nearest handler of a given type in the call stack, that is:

- the handler whose try block was most recently entered and not yet exited,
- the handler whose type matches the raise expression.

A handler of T exception type is a match for a raise expression with an exception type of E if:

- T and E are of the same type, or
- T is a supertype of E.

In order to handle system signals as exceptions, make sure to insert macro OCC_CATCH_SIGNALS somewhere in the beginning of the relevant code. The recommended location for it is first statement after opening brace of try {} block.

As an example, consider the exceptions of type NumericError, Overflow, Underflow and ZeroDivide where NumericError is the supertype of the three others.

Example

```
void f(1)
{
  try {
    OCC_CATCH_SIGNALS
    // try block
  }
  catch(Standard_Overflow) { // first handler
    // ...
  }
  catch(Standard_NumericError) { // second handler
    // ...
  }
}
```

Here, the first handler will catch exceptions of Overflow type and the second one – exceptions of NumericError type and all exceptions derived from it, including Underflow and Zerodivide.

The handlers are checked in order of appearance, from the nearest to the most distant try block, until one matches the raise expression. For a try block, it would be a mistake to place a handler for a base exception type ahead of a handler for its derived type since that would ensure that the handler for the derived exception would never be invoked.

Example

```
void f(1)
{
  int i = 0;

  {
    try {
      OCC_CATCH_SIGNALS
      g(i);// i is accessible
    }
    // statement here will produce compile-time errors !
    catch(Standard_NumericError) {
      // fix up with possible reuse of i
    }
    // statement here may produce unexpected side effect
  }
  . . .
}
```

The exceptions form a hierarchy tree completely separated from other user defined classes. One exception of type Failure is the root of the entire exception hierarchy. Thus, using a handler with Failure type catches any OCCT exception. It is recommended to set up such a handler in the main routine.

The main routine of a program would look like this:

Example

```
#include <Standard_ErrorHandler.hxx>
#include <Standard_Failure.hxx>
#include <iostream.h>
int main (int argc, char* argv[])
{
  try {
    OCC_CATCH_SIGNALS
```

```
    // main block
    return 0;
  }
  catch(Standard_Failure) {
    Handle(Standard_Failure) error = Standard_Failure::Caught ();
    cout << error << end1;
  }
  return 1;
}
```

where the function Caught is a static member of Failure that returns an exception object containing the error message built in the raise expression. Note that this method of accessing a raised object is used in Open CASCADE Technology instead of usual C++ syntax (receiving the exception in catch argument).

NOTE

Though standard C++ scoping rules and syntax apply to try block and handlers, note that on some platforms Open CASCADE Technology may be compiled in compatibility mode when exceptions are emulated by long jumps (see below). In this mode it is required that no statement precedes or follows any handler. Thus it is highly recommended to always include a try block into additional {} braces. Also this mode requires that header file Standard_ErrorHandler.hxx be included in your program before a try block, otherwise it may fail to handle Open CASCADE Technology exceptions; furthermore catch() statement does not allow passing exception object as argument.

Catching signals

In order for the application to be able to catch system signals (access violation, division by zero, etc.) in the same way as other exceptions, the appropriate signal handler shall be installed in the runtime by the method

OSD::SetSignal();

Normally this method is called in the beginning of the main() function. It installs a handler that will convert system signals into OCCT exceptions.

In order to actually convert signals to exceptions, macro OCC_CATCH_SIGNALS shall be inserted in the source code. The typical place where this macro is put is beginning of the try{} block which catches such exceptions.

2.4.3. Implementation details

The exception handling mechanism in Open CASCADE Technology is implemented in different ways depending on the preprocessor macros NO_CXX_EXCEPTIONS and OCC_CONVERT_SIGNALS, which shall be consistently defined by compilation procedures for both Open CASCADE Technology and user applications:

1. On Windows and DEC, these macros are not defined by default, and normal C++ exceptions are used in all cases, including throwing from signal handler. Thus the behavior is as expected in C++.

2. On SUN and Linux, macro OCC_CONVERT_SIGNALS is defined by default. The C++ exception mechanism is used for catching exceptions and for throwing them from normal code. Since it is not possible to throw C++ exception from system signal handler function, that function makes a long jump to the nearest (in the execution stack) invocation of macro OCC_CATCH_SIGNALS, and only there the C++ exception gets actually thrown. The macro OCC_CATCH_SIGNALS is defined in the file Standard_ErrorHandler.hxx. Therefore, including this file is necessary for successful compilation of a code containing this macro.

   This mode differs from standard C++ exception handling only for signals:

   - macro OCC_CATCH_SIGNALS is necessary (besides call to OSD::SetSignal() described above) for conversion of signals into exceptions;
   - the destructors for automatic C++ objects created in the code after that macro and till the place where signal is raised will not be called in case of signal, since no C++ stack unwinding is performed by long jump.

3. On SUN and Linux Open CASCADE Technology can also be compiled in compatibility mode (which was default till Open CASCADE Technology 6.1.0). In that case macro NO_CXX_EXCEPTIONS is defined and the C++ exceptions are simulated with C long jumps. As a consequence, the behavior is slightly different from that expected in the C++ standard.

   While exception handling with NO_CXX_EXCEPTIONS is made very similar to C++ by syntax, it has a number of peculiarities that should be taken into account:

   - try and catch are actually macros defined in the file Standard_ErrorHandler.hxx. Therefore, including this file is necessary for handling OCCT exceptions;
   - due to being a macro, catch cannot contain a declaration of the exception object after its type; only type is allowed in the catch statement. Use method Standard_Failure::Caught() to access an exception object;
   - catch macro may conflict with some STL classes that might use catch(…) statements in their header files. So STL headers should not be included after Standard_ErrorHandler.hxx;
   - Open CASCADE Technology try/catch block will not handle normal C++ exceptions; however this can be achieved using special workarounds;
   - the try macro defines a C++ object that holds an entry point in the exception handler. Therefore if exception is raised by code located immediately after the try/catch block but on the same nesting level as *try*, it may be handled by that *catch*. This may lead to unexpected behavior, including infinite loop. To avoid that, always surround the try/catch block in {} braces;
   - the destructors of the C++ objects allocated on the stack after handler initialization are not called by exception raising.

In general, for writing platform-independent code it is recommended to insert macros OCC_CATCH_SIGNALS in try {} blocks or other code where signals may happen. For compatibility with previous versions of Open CASCADE Technology the limitations described above for NO_CXX_EXCEPTIONS shall be assumed.

2.5. Plug-In Management

2.5.1. Distribution by Plug-Ins

A plug-in is a component that can be loaded dynamically into a client application, not requiring to be directly linked to it. The plug-in is not bound to its client, i.e. the plug-in knows only how its connection mechanism is defined and how to call the corresponding services.

A plug-in can be used to:

- implement the mechanism of a *driver*, i.e dynamically changing a driver implementation according to the current transactions (for example, retrieving a document stored in another version of an application),
- restrict processing resources to the minimum required (for example, it does not load any application services at run-time as long as the user does not need them),
- facilitate development de-synchronization (an application can be delivered with base functions while some advanced capabilities will be added as plug-ins when they are available).

The plug-in is identified with the help of the global universal identifier (GUID). The GUID includes lower case characters and cannot end with a blank space.

Once it has been loaded, the call to the services provided by the plug-in is direct (the client is implemented in the same language as the plug-in).

C++ Plug-In Implementation

The C++ plug-in implements a service as an object with functions defined in an abstract class (this abstract class and its parent classes with the GUID are the only information about the plug-in implemented in the client application). The plug-in consists of a sharable library including a method named Factory which creates the C++ object (the client cannot instantiate this object because the plug-in implementation is not visible).

Foundation classes provide in the package **Plugin** a method named Load(), which enables the client to access the required service through a library.

That method reads the information regarding available plug-ins and their locations from the resource file Plugin found by environment variable CSF_PluginDefaults:

$CSF_PluginDefaults/.Plugin

The Load method:

- looks for the library name in the resource file or registry through its GUID,

---

Example(UNIX)

```
! METADATADRIVER whose value must be OS or DM.

! FW
a148e300-5740-11d1-a904-080036aaa103.Location:
```

```
libFWOSPlugin.so
a148e300-5740-11d1-a904-080036aaa103.CCL:
/adv_44/CAS/BAG/FW-K4C/inc/FWOS.ccl

! FWDM
a148e301-5740-11d1-a904-080036aaa103.Location:
libFWDMPlugin.so
a148e301-5740-11d1-a904-080036aaa103.CCL:
/adv_44/CAS/BAG/DESIGNMANAGER-K4C/inc/DMAccess.ccl|/
adv_44/CAS/BAG/DATABASE-K4C/inc/FWDMCommands.ccl
a148e301-5740-11d1-a904-080036aaa103.Message: /adv_44/CAS/
BAG/DESIGNMANAGER-K4C/etc/locale/DMAccess

! Copy-Paste
5ff7dc00-8840-11d1-b5c2-00a0c9064368.Location:
libCDMShapeDriversPlugin.so
5ff7dc01-8840-11d1-b5c2-00a0c9064368.Location:
libCDMShapeDriversPlugin.so
5ff7dc02-8840-11d1-b5c2-00a0c9064368.Location:
libCDMShapeDriversPlugin.so
5ff7dc03-8840-11d1-b5c2-00a0c9064368.Location:
libCDMShapeDriversPlugin.so
5ff7dc04-8840-11d1-b5c2-00a0c9064368.Location:
libCDMShapeDriversPlugin.so

! Plugs 2d plotters:
d0d722a2-b4c9-11d1-b561-0000f87a4710.location: FWOSPlugin
d0d722a2-b4c9-11d1-b561-0000f87a4710.CCL: /adv_44/CAS/BAG/
VIEWERS-K4C/inc/CCLPlotters.ccl
d0d722a2-b4c9-11d1-b561-0000f87a4710.Message: /adv_44/CAS/
BAG/VIEWERS-K4C/etc/locale/CCLPlotters

!SHAPES
e3708f72-b1a8-11d0-91c2-080036424703.Location:
libBRepExchangerPlugin.so
e3708f72-b1a8-11d0-91c2-080036424703.CCL: /adv_44/CAS/BAG/
FW-K4C/inc/BRep.ccl
```

- loads the library according to the rules of the operating system of the host machine (for example, by using environment variables such as LD_LIBRARY_PATH with Unix and PATH with Windows), then,
- invokes the Factory method to return the object which supports the required service.

The client may then call the functions supported by this object.

C++ Client Plug-In Implementation

To invoke one of the services provided by the plug-in, you may call the Plugin::ServiceFactory global function with the Standard_GUID of the requested service as follows:

```
Handle(FADriver_PartStorer)::DownCast
  (PlugIn::ServiceFactory
      (PlugIn_ServiceId(yourStandardGUID)))
```

Example// File:FAFactory.cxx

```
#include <FAFactory.ixx>

#include <FADriver_PartRetriever.hxx>
#include <FADriver_PartStorer.hxx>
#include <FirstAppSchema.hxx>
#include <Standard_GUID.hxx>
#include <Standard_Failure.hxx>
#include <FACDM_Application.hxx>
#include <Plugin_Macro.hxx>

PLUGIN(FAFactory)

static Standard_GUID
       StorageDriver("45b3c690-22f3-11d2-b09e-0000f8791463");
static Standard_GUID
       RetrievalDriver("45b3c69c-22f3-11d2-b09e-0000f8791463");
static Standard_GUID
       Schema("45b3c6a2-22f3-11d2-b09e-0000f8791463");

//=======================================================
// function : Factory
// purpose :
//=======================================================
Handle(Standard_Transient) FAFactory::Factory(const Standard_GUID& aGUID)
{
  if(aGUID == StorageDriver) {
    cout << "FAFactory : Create store driver" << endl;
    static Handle(FADriver_PartStorer) sd = new FADriver_PartStorer();
    return sd;
  }

  if(aGUID == RetrievalDriver) {
    cout << "FAFactory : Create retrieve driver" << endl;
```

```
    static Handle(FADriver_PartRetriever)
    rd = new FADriver_PartRetriever();
    return rd;
  }

  if(aGUID == Schema) {
    cout << “FAFactory : Create schema” << endl;
    static Handle(FirstAppSchema) s = new FirstAppSchema();
    return s;
  }

  Standard_Failure::Raise(“FAFactory: unknown GUID”);
  Handle(Standard_Transient) t;
  return t;
}
```

Not Using the Software Factory

To create a factory without using the Software Factory, define a **dll** project under Windows or a library under UNIX by using a source file as specified above. The **FAFactory** class is implemented as follows:

Example

```
#include <Handle_Standard_Transient.hxx>
#include <Standard_Macro.hxx>
class Standard_Transient;
class Standard_GUID;
class FAFactory {
public:
  Standard_EXPORT static Handle_Standard_Transient
                  Factory(const Standard_GUID& aGUID) ;
  . . .
};
```

3. Collections, Strings and Unit Conversion

3.1. Collections

3.1.1. Overview

The **Collections** component contains the classes that handle dynamically sized aggregates of data. They include a wide range of collections such as arrays, lists and maps.

Collections classes are *generic*, that is, they can hold a variety of objects which do not necessarily inherit from a unique root class. When you need to use a collection of a given type of object you must *instantiate* it for this specific type of element. Once this declaration is compiled, all the functions available on the generic collection are available on your *instantiated class*. Note however:

- Each collection directly used as an argument in OCCT public syntax is instantiated in an OCCT component.
- The **TColStd** package (**Collections of Standard Objects** component) provides numerous instantiations of these generic collections with objects from the **Standard** package or from the **Strings** component.

The **Collections** component provides a wide range of generic collections:

- *Arrays* are generally used for a quick access to the item, however an array is a fixed sized aggregate.
- *Sequences* are variable sized structures, they avoid the use of large and quasi-empty arrays. But a sequence item is longer to access than an array item: only an exploration in sequence is effective (but sequences are not adapted for numerous explorations). Arrays and sequences are commonly used as data structures for more complex objects.
- On the other hand, *maps* are dynamic structures where the size is constantly adapted to the number of inserted items and the access time for an item is effective. Maps structures are commonly used in cases of numerous explorations: they are typically internal data structures for complex algorithms. *Sets* generate the same results as maps but computation time is considerable.
- *Lists, queues* and *stacks* are minor structures similar to sequences but with other exploration algorithms.

Most collections follow *value semantics*: their instances are the actual collections, not *handles* to a collection. Only arrays and sequences may also be manipulated by handle, and therefore shared.

Generic general-purpose aggregates (TCollection package)

Array1

Array2

HArray1

HArray2

HSequence

HSet

List

Queue

Sequence

Set

Stack

Generic maps (TCollection package)

BasicMap

DataMap

DoubleMap

IndexedDataMap

IndexedMap

Map

MapHasher

Iterators (TCollection package)

BasicMapIterator

DataMapIterator

DoubleMapIterator

ListIterator

MapIterator

SetIterator

StackIterator

3.1.2. Generic general-purpose Aggregates

TCollection_Array1

Unidimensional arrays similar to C arrays, i.e. of fixed size but dynamically dimensioned at construction time.

As with a C array, the access time for an **Array1** indexed item is constant and is independent of the array size. Arrays are commonly used as elementary data structures for more complex objects.

Array1 is a generic class which depends on **Item**, the type of element in the array.

Array1 indexes start and end at a user-defined position. Thus, when accessing an item, you must base the index on the lower and upper bounds of the array.

TCollection_Array2

Bi-dimensional arrays of fixed size but dynamically dimensioned at construction time.

As with a C array, the access time for an **Array2** indexed item is constant and is independent of the array size. Arrays are commonly used as elementary data structures for more complex objects.

Array2 is a generic class which depends on **Item**, the type of element in the array.

Array2 indexes start and end at a user-defined position. Thus, when accessing an item, you must base the index on the lower and upper bounds of the array.

TCollection_HArray1

Unidimensional arrays similar to C arrays, i.e. of fixed size but dynamically dimensioned at construction time.

As with a C array, the access time for an **HArray1** or **HArray2** indexed item is constant and is independent of the array size. Arrays are commonly used as elementary data structures for more complex objects.

HArray1 objects are *handles* to arrays.

- **HArray1** arrays may be shared by several objects.
- You may use a **TCollection_Array1** structure to have the actual array.

HArray1 is a generic class which depends on two parameters:

- **Item**, the type of element in the array,
- **Array**, the actual type of array handled by **HArray1**. This is an instantiation with **Item** of the **TCollection_Array1** generic class.

HArray1 indexes start and end at a user-defined position. Thus, when accessing an item, you must base the index on the lower and upper bounds of the array.

TCollection_HArray2

Bi-dimensional arrays of fixed size but dynamically dimensioned at construction time.

As with a C array, the access time for an **HArray2** indexed item is constant and is independent of the array size. Arrays are commonly used as elementary data structures for more complex objects.

HArray2 objects are *handles* to arrays.

- **HArray2** arrays may be shared by several objects.
- You may use a **TCollection_Array2** structure to have the actual array.

HArray2 is a generic class which depends on two parameters:

- **Item**, the type of element in the array,
- **Array**, the actual type of array handled by **HArray2**. This is an instantiation with **Item** of the **TCollection_Array2** generic class.

TCollection_HSequence

A sequence of items indexed by an integer.

Sequences have about the same goal as unidimensional arrays (**TCollection_HArray1**): they are commonly used as elementary data structures for more complex objects. But a sequence is a structure of *variable size*: sequences avoid the use of large and quasi-empty arrays. Exploring a sequence data structure is effective when the exploration is done *in sequence;* elsewhere a sequence item is longer to read than an array item. Note also that sequences are not effective when they have to support numerous algorithmic explorations: a map is better for that.

HSequence objects are *handles* to sequences.

- **HSequence** sequences may be shared by several objects.
- You may use a **TCollection_Sequence** structure to have the actual sequence.

HSequence is a generic class which depends on two parameters:

- **Item**, the type of element in the sequence,
- **Seq**, the actual type of sequence handled by **HSequence**. This is an instantiation with **Item** of the **TCollection_Sequence** generic class.

TCollection_HSet

Collection of non-ordered items without any duplicates. At each transaction, the system ckecks to see that there are no duplicates.

HSet objects are *handles* to sets.

HSet is a generic class which depends on two parameters:

- **Item**, the type of element in the set,
- **Set**, the actual type of set handled by **HSet**. This is an instantiation with **TCollection_Set** generic class.

TCollection_List

Ordered lists of non-unique objects which can be accessed sequentially using an iterator.

Item insertion in a list is very fast at any position. But searching for items by value may be slow if the list is long, because it requires a sequential search.

List is a generic class which depends on **Item**, the type of element in the structure.

Use a **ListIterator** iterator to explore a **List** structure.

An iterator class is automatically instantiated from the **TCollection_ListIterator** class at the time of instantiation of a **List** structure.

A sequence is a better structure when searching for items by value.

Queues and stacks are other kinds of list with a different access to data.

TCollection_Queue

A structure where items are added at the end and removed from the front. The first item entered will be the first removed ("**FIFO**" structure: First In First Out). **Queue** is a generic class which depends on **Item**, the type of element in the structure.

TCollection_Sequence

A sequence of items indexed by an integer.

Sequences have about the same goal as unidimensional arrays (**TCollection_Array1**): they are commonly used as elementary data structures for more complex objects. But a sequence is a structure of *variable size*: sequences avoid the use of large and quasi-empty arrays. Exploring a sequence data structure is effective when the exploration is done *in sequence*; elsewhere a sequence item is longer to read than an array item. Note also that sequences are not effective when they have to support numerous algorithmic explorations: a map is better for that.

Sequence is a generic class which depends on **Item**, the type of element in the sequence.

TCollection_Set

Collection of non-ordered items without any duplicates. At each transaction, the system ckecks there are no duplicates.

A set generates the same result as a map. A map is more effective; so it is advisable to use maps instead of sets.

Set is a generic class which depends on **Item**, the type of element in the set.

Use a **SetIterator** iterator to explore a **Set** structure.

TCollection_Stack

A structure where items are added and removed from the top. The last item entered will be the first removed ("**LIFO**" structure: Last In First Out).

Stack is a generic class which depends on **Item**, the type of element in the structure.

Use a **StackIterator** iterator to explore a **Stack** structure.

3.1.3. Generic Maps

TCollection_BasicMap

Root class for maps.

Maps are dynamically extended data structures where data is quickly accessed with a key.

General properties of maps

Map items may be (complex) non-unitary data; they may be difficult to manage with an array. More, the map allows a data structure to be indexed by complex data.

The size of a map is dynamically extended. So a map may be first dimensioned for a little number of items. Maps avoid the use of large and quasi-empty arrays. The access time for a map item is much better than the one for a sequence, list, queue or stack item.

The access time for a map item may be compared with the access time for an array item. It depends first on the size of the map. It depends also on the quality of a user redefinable function (the *hashing function*) to find quickly where is the item.

The exploration of a map may be of better performance than the exploration of an array because the size of the map is adapted to the number of inserted items.

These properties explain why maps are commonly used as internal data structures for algorithms.

Definitions

A map is a data structure for which data are addressed by *keys*.

Once inserted in the map, a map item is referenced as an *entry* of the map.

Each entry of the map is addressed by a key. Two different keys address two different entries of the map.

The position of an entry in the map is called a *bucket*.

A map is dimensioned by its number of buckets, i.e. the maximum number of entries in the map. The performance of a map is conditioned by the number of buckets.

The *hashing function* transforms a key into a bucket index. The number of values that can be computed by the hashing function is equal to the number of buckets of the map.

Both the hashing function and the equality test between two keys are provided by a *hasher* object.

A map may be explored by a *map iterator*. This exploration provides only inserted entries in the map (i.e. non empty buckets).

Collections generic maps

The **Collections** component provides numerous generic derived maps.

These maps include automatic management of the number of *buckets*: they are automatically resized when the number of *keys* exceeds the number of buckets. If you have a fair idea of the number of items in your map, you can save on automatic resizing by specifying a number of buckets at the time of construction, or by using a resizing function. This may be considered for crucial optimization issues.

Keys, items and *hashers* are parameters of these generic derived maps.

TCollection_MapHasher class describes the functions required by any *hasher* which is to be used with a map instantiated from the **Collections** component.

An iterator class is automatically instantiated at the time of instantiation of a map provided by the **Collections** component if this map is to be explored with an iterator. Note that some provided generic maps are not to be explored with an iterator but with indexes (*indexed maps*).

TCollection_DataMap

A map used to store keys with associated items. An entry of **DataMap** is composed of both the key and the item.

The **DataMap** can be seen as an extended array where the keys are the indexes.

DataMap is a generic class which depends on three parameters:

- **Key** is the type of key for an entry in the map,
- **Item** is the type of element associated with a key in the map,
- **Hasher** is the type of hasher on keys.

Use a **DataMapIterator** iterator to explore a DataMap map.

An iterator class is automatically instantiated from the

TCollection_DataMapIterator generic class at the time of instantiation of a **DataMap** map.

TCollection_MapHasher class describes the functions required for a **Hasher** object.

TCollection_DoubleMap

A map used to bind pairs of keys (Key1,Key2) and retrieve them in linear time.

Key1 is referenced as the *first key* of the **DoubleMap** and Key2 as the *second key*.

An entry of a **DoubleMap** is composed of a pair of two keys: the first key and the second key.

DoubleMap is a generic class which depends on four parameters:

- **Key1** is the type of the first key for an entry in the map,
- **Key2** is the type of the second key for an entry in the map,
- **Hasher1** is the type of hasher on first keys,
- **Hasher2** is the type of hasher on second keys.

Use a **DoubleMapIterator** to explore a **DoubleMap** map.

An iterator class is automatically instantiated from the **TCollection_DoubleMapIterator** class at the time of instantiation of a **DoubleMap** map.

TCollection_MapHasher class describes the functions required for a **Hasher1** or a **Hasher2** object.

TCollection_IndexedDataMap

A map to store keys with associated items and to bind an index to them.

Each new key stored in the map is assigned an index. Indexes are incremented as keys (and items) stored in the map. A key can be found by the index, and an index can be found by the key. No key but the last can be removed, so the indexes are in the range 1...Upper where Upper is the number of keys stored in the map. An item is stored with each key.

An entry of an **IndexedDataMap** is composed of both the key, the item and the index. An **IndexedDataMap** is an ordered map, which allows a linear iteration on its contents. It combines the interest:

- of an array because data may be accessed with an index,
- and of a map because data may also be accessed with a key.

IndexedDataMap is a generic class which depends on three parameters:

- **Key** is the type of key for an entry in the map,
- **Item** is the type of element associated with a key in the map,
- **Hasher** is the type of hasher on keys.

TCollection_IndexedMap

A map used to store keys and to bind an index to them.

Each new key stored in the map is assigned an index. Indexes are incremented as keys stored in the map. A key can be found by the index, and an index by the key. No key but the last can be removed, so the indexes are in the range 1...Upper where Upper is the number of keys stored in the map.

An entry of an **IndexedMap** is composed of both the key and the index. An **IndexedMap** is an ordered map, which allows a linear iteration on its contents. But no data is attached to the key. An **IndexedMap** is typically used by an algorithm to know if some action is still performed on components of a complex data structure.

IndexedMap is a generic class which depends on two parameters:

- **Key** is the type of key for an entry in the map,
- **Hasher** is the type of hasher on keys.

TCollection_Map

A basic hashed map, used to store and retrieve keys in linear time.

An entry of a **Map** is composed of the key only. No data is attached to the key. A **Map** is typically used by an algorithm to know if some action is still performed on components of a complex data structure.

Map is a generic class which depends on two parameters:

- **Key** is the type of key in the map,
- **Hasher** is the type of hasher on keys.

Use a **MapIterator** iterator to explore a **Map** map.

TCollection_MapHasher

A hasher on the *keys* of a map instantiated from the **Collections** component.

A hasher provides two functions:

- The *hashing function* (**HashCode**) transforms a *key* into a *bucket* index in the map. The number of values that can be computed by the hashing function is equal to the number of buckets in the map.
- **IsEqual** is the equality test between two keys. Hashers are used as parameters in generic maps provided by the **Collections** component.

MapHasher is a generic class which depends on the type of keys, providing that **Key** is a type from the **Standard** package. In such cases **MapHasher** may be directly instantiated with **Key**. Note that the package **TColStd** provides some of these instantiations.

Elsewhere, if **Key** is not a type from the **Standard** package you must consider **MapHasher** as a template and build a class which includes its functions, in order to use it as a hasher in a map instantiated from the **Collections** component.

Note that **TCollection_AsciiString** and **TCollection_ExtendedString** classes correspond to these specifications, in consequence they may be used as hashers: when **Key** is one of these two types you may just define the hasher as the same type at the time of instantiation of your map.

3.1.4. Iterators

TCollection_BasicMapIterator

Root class for map iterators. A map iterator provides a step by step exploration of all the entries of a map.

TCollection_DataMapIterator

Functions used for iterating the contents of a **DataMap** map.

A map is a non-ordered data structure. The order in which entries of a map are explored by the iterator depends on its contents and change when the map is edited.

It is not recommended to modify the contents of a map during the iteration: the result is unpredictable.

TCollection_DoubleMapIterator

Functions used for iterating the contents of a **DoubleMap** map.

TCollection_ListIterator

Functions used for iterating the contents of a **List** data structure.

A **ListIterator** object can be used to go through a list sequentially, and as a bookmark to hold a position in a list. It is not an index, however. Each step of the iteration gives the *current position* of the iterator, to which corresponds the *current item* in the list. The *current position* is *undefined* if the list is empty, or when the exploration is finished.

An iterator class is automatically instantiated from this generic class at the time of instantiation of a **List** data structure.

TCollection_MapIterator

Functions used for iterating the contents of a **Map** map.

An iterator class is automatically instantiated from this generic class at the time of instantiation of a **Map** map.

TCollection_SetIterator

Functions used for iterating the contents of a **Set** data structure.

An iterator class is automatically instantiated from this generic class at the time of instantiation of a **Set** structure.

TCollection_StackIterator

Functions used for iterating the contents of a **Stack** data structure.

An iterator class is automatically instantiated from this generic class at the time of instantiation of a **Stack** structure.

3.2. Collections of Standard Objects

3.2.1. Overview

While generic classes of the **TCollection** package are the root classes that describe the generic purpose of every type of collection, classes effectively used are extracted from the **TColStd** package.

The **TColStd** and **TShort** packages provide frequently used instantiations of generic classes with objects from the **Standard** package or strings from the **TCollection** package.

3.2.2. Description

These instantiations are the following:

- Unidimensional arrays: instantiations of the **TCollection_Array1** generic class with **Standard** Objects and **TCollection** strings.
- Bidimensional arrays: instantiations of the **TCollection_Array2** generic class with **Standard** Objects.
- Unidimensional arrays manipulated by handles: instantiations of the **TCollection_HArray1** generic class with **Standard** Objects and **TCollection** strings.
- Bidimensional arrays manipulated by handles: instantiations of the **TCollection_HArray2** generic class with **Standard** Objects.
- Sequences: instantiations of the **TCollection_Sequence** generic class with **Standard** objects and **TCollection** strings.
- Sequences manipulated by handles: instantiations of the **TCollection_HSequence** generic class with **Standard** objects and **TCollection** strings.
- Lists: instantiations of the **TCollection_List** generic class with **Standard** objects.
- Queues: instantiations of the **TCollection_Queue** generic class with **Standard** objects.
- Sets: instantiations of the **TCollection_Set** generic class with **Standard** objects.
- Sets manipulated by handles: instantiations of the **TCollection_HSet** generic class with **Standard** objects.
- Stacks: instantiations of the **TCollection_Stack** generic class with **Standard** objects.
- Hashers on map keys: instantiations of the **TCollection_MapHasher** generic class with **Standard** objects.
- Basic hashed maps: instantiations of the **TCollection_Map** generic class with **Standard** objects.
- Hashed maps with an additional item: instantiations of the **TCollection_DataMap** generic class with **Standard** objects.
- Basic indexed maps: instantiations of the **TCollection_IndexedMap** generic class with **Standard** objects.
- Indexed maps with an additional item: instantiations of the **TCollection_IndexedDataMap** generic class with **Standard_Transient** objects.

- Class **TColStd_PackedMapOfInteger** provides alternative implementation of map of integer numbers, optimized for both performance and memory usage (it uses bit flags to encode integers, which results in spending only 24 bytes per 32 integers stored in optimal case). This class also provides Boolean operations with maps as sets of integers (union, intersection, subtraction, difference, checks for equality and containment).

3.3. Strings

3.3.1. Overview

The **Strings** component provides services to manipulate character strings.

Strings are classes that handle dynamically sized sequences of characters based on both ASCII (normal 8-bit character type) and Unicode (16-bit character type). They provide editing operations with built-in memory management which make the relative objects easier to use than ordinary character arrays.

Strings may also be manipulated by *handle*, and therefore shared.

Strings (TCollection package)

AsciiString

ExtendedString

HAsciiString

HExtendedString

Conversion (Resource package)

Unicode

3.3.2. Strings

TCollection_AsciiString

A variable-length sequence of ASCII characters (normal 8-bit character type). It provides editing operations with built-in memory management to make **AsciiString** objects easier to use than ordinary character arrays.

AsciiString objects follow "value semantics", that is, they are the actual strings, not handles to strings, and are copied through assignment. You may use **HAsciiString** objects to get handles to strings.

TCollection_ExtendedString

A variable-length sequence of "extended" (UNICODE) characters (16-bit character type). It provides editing operations with built-in memory management to make **ExtendedString** objects easier to use than ordinary extended character arrays.

ExtendedString objects follow "value semantics", that is, they are the actual strings, not handles to strings, and are copied through assignment. You may use **HExtendedString** objects to get handles to strings.

TCollection_HAsciiString

A variable-length sequence of ASCII characters (normal 8-bit character type). It provides editing operations with built-in memory management to make **HAsciiString** objects easier to use than ordinary character arrays.

HAsciiString objects are *handles* to strings.

- **HAsciiString** strings may be shared by several objects.
- You may use an **AsciiString** object to get the actual string.

HAsciiString objects use an **AsciiString** string as a field.

TCollection_HExtendedString

A variable-length sequence of "extended" (UNICODE) characters (16-bit character type). It provides editing operations with built-in memory management to make **ExtendedString** objects easier to use than ordinary extended character arrays.

HExtendedString objects are *handles* to strings.

- **HExtendedString** strings may be shared by several objects.
- You may use an **ExtendedString** object to get the actual string.

HExtendedString objects use an **ExtendedString** string as a field.

3.3.3. Conversion

Resource_Unicode

Functions used to convert a non-ASCII *C string* given in ANSI, EUC, GB or SJIS format, to a Unicode string of extended characters, and vice versa.

3.4. Unit Conversion

3.4.1. Overview

The **UnitsAPI** global functions are used to convert a value from any unit into another unit. Conversion is executed among three unit systems:

- the **SI System**,
- the user's **Local System**,
- the user's **Current System**.

The **SI System** is the standard international unit system. It is indicated by *SI* in the signatures of the **UnitsAPI** functions.

The OCCT (former MDTV) System corresponds to the SI international standard but *the length unit and all its derivatives use the millimeter* instead of the meter.

Both systems are proposed by Open CASCADE Technology; the SI System is the standard option. By selecting one of these two systems, you define your **Local System** through the **SetLocalSystem** function. The **Local System** is indicated by *LS* in the signatures of the **UnitsAPI** functions.

The Local System units can be modified in the working environment. You define your **Current System** by modifying its units through the **SetCurrentUnit** function. The Current System is indicated by *Current* in the signatures of the **UnitsAPI** functions.

A physical quantity is defined by a string (example: LENGTH).

4. Math Primitives and Algorithms

4.1. Overview

Math primitives and algorithms available in Open CASCADE Technology include:

- Vectors and matrices
- Geometric primitives
- Math algorithms

4.2. Vectors and Matrices

The Vectors and Matrices component provides a C++ implementation of the fundamental types Matrix and Vector, currently used to define more complex data structures. The Vector and Matrix classes support vectors and matrices of real values with standard operations such as addition, multiplication, transposition, inversion etc.

Vectors and matrices have arbitrary ranges which must be defined at declaration time and cannot be changed after declaration.

Example

```
math_Vector v(1, 3);
// a vector of dimension 3 with range (1..3)

math_Matrix m(0, 2, 0, 2);
// a matrix of dimension 3x3 with range (0..2, 0..2)

math_Vector v(N1, N2);
// a vector of dimension N2-N1+1 with range (N1..N2)
```

Vector and Matrix objects use value semantics. In other words, they cannot be shared and are copied through assignment.

Example

```
math_Vector v1(1, 3), v2(0, 2);

v2 = v1;
```

```
// v1 is copied into v2. a modification of v1 does not
//affect v2
```

Vector and Matrix values may be initialized and obtained using indexes which must lie within the range definition of the vector or the matrix.

Example

```
math_Vector v(1, 3);
math_Matrix m(1, 3, 1, 3);
Standard_Real value;

v(2) = 1.0;
value = v(1);
m(1, 3) = 1.0;
value = m(2, 2);
```

Some operations on Vector and Matrix objects may not be legal. In this case an exception is raised. Two standard exceptions are used:

- Standard_DimensionError exception is raised when two matrices or vectors involved in an operation are of incompatible dimensions.
- Standard_RangeError exception is raised if an access outside the range definition of a vector or of a matrix is attempted.

Example

```
math_Vector v1(1, 3), v2(1, 2), v3(0, 2);
v1 = v2;
// error: Standard_DimensionError is raised

v1 = v3;
// OK: ranges are not equal but dimensions are
// compatible

v1(0) = 2.0;
// error: Standard_RangeError is raised
```

4.3. Primitive Geometric Types

4.3.1. Overview

Before creating a geometric object, you must decide whether you are in a 2d or in a 3d context and how you want to handle the object.

The *gp* package offers classes for both 2d and 3d objects which are handled by value rather than by reference. When this sort of object is copied, it is copied entirely. Changes in one instance will not be reflected in another.

4.3.2. gp

The *gp* package defines the basic non-persistent geometric entities used for algebraic calculation and basic analytical geometry in 2d & 3d space. It also provides basic transformations such as identity, rotation, translation, mirroring, scale transformations, combinations of transformations, etc. Entities are handled by value.

The available geometric entities are:

- 2d & 3d Cartesian coordinates (x, y, z)
- Matrices
- Cartesian points
- Vector
- Direction
- Axis
- Line
- Circle
- Ellipse
- Hyperbola
- Parabola
- Plane
- Infinite cylindrical surface
- Spherical surface
- Toroidal surface
- Conical surface.

4.4. Collections of Primitive Geometric Types

Before creating a geometric object, you must decide whether you are in a 2d or in a 3d context and how you want to handle the object.

If you do not need a single instance of a geometric primitive but a set of them then the package which deals with collections of this sort of object, *TColgp*, will provide the necessary functionality.

In particular, this package provides standard and frequently used instantiations of generic classes with geometric objects.

4.4.1. TColgp

The *TColgp* package provides instantiations of the TCollection classes with the classes from *gp* i.e. *XY*, *XYZ*, *Pnt*, *Pnt2d*, *Vec*, *Vec2d*, *Lin*, *Lin2d*, *Circ*, *Circ2d.*

These are non-persistent classes.

4.5. Basic Geometric Libraries

There are various library packages available which offer a range of basic computations on curves and surfaces.

If you are dealing with objects created from the *gp* package, the useful algorithms are in the elementary curves and surfaces libraries - the *ElCLib* and *ElSLib* packages.

The *Precision* package describes functions for defining the precision criterion used to compare two numbers.

4.5.1. ElCLib

Methods for analytic curves. A library of simple computations on curves from the *gp* package (Lines, Circles and Conics). Computes points with a given parameter. Computes the parameter for a point.

4.5.2. ElSLib

Methods for analytic surfaces . A library of simple computations on surfaces from the package *gp* (Planes, Cylinders, Spheres, Cones, Tori). Computes points with a given pair of parameters. Computes the parameter for a point. There is a library for calculating normals on curves and surfaces.

4.5.3. Bnd

Package Bnd provides a set of classes and tools to operate with bounding boxes of geometric objects in 2d and 3d space.

4.6. Common Math Algorithms

The common math algorithms library provides a C++ implementation of the most frequently used mathematical algorithms. These include:

- Algorithms to solve a set of linear algebraic equations,
- Algorithms to find the minimum of a function of one or more independent variables,
- Algorithms to find roots of one, or of a set, of non-linear equations,
- An algorithm to find the eigenvalues and eigenvectors of a square matrix.

4.6.1. Implementation of Algorithms

All mathematical algorithms are implemented using the same principles. They contain:

A constructor performing all, or most of, the calculation, given the appropriate arguments. All relevant information is stored inside the resulting object, so that all subsequent calculations or interrogations will be solved in the most efficient way.

A function IsDone returning the boolean true if the calculation was successful.

A set of functions, specific to each algorithm, enabling all the various results to be obtained.

Calling these functions is legal only if the function IsDone answers true, otherwise the exception StdFail_NotDone is raised.

The example below demonstrates the use of the Gauss class, which implements the Gauss solution for a set of linear equations.The following definition is an extract from the header file of the class math_Gauss:

Example

```
class Gauss {
public:
  Gauss (const math_Matrix& A);
  Standard_Boolean IsDone() const;
  void Solve (const math_Vector& B,
  math_Vector& X) const;
};
```

Now the main program uses the Gauss class to solve the equations a*x1=b1 and a*x2=b2:

Example

```
#include <math_Vector.hxx> #include <math_Matrix.hxx>
main ()
{
  math_Vector a(1, 3, 1, 3);
  math_Vector b1(1, 3), b2(1, 3);
  math_Vector x1(1, 3), x2(1, 3);
  // a, b1 and b2 are set here to the appropriate values
  math_Gauss sol(a);          // computation of the
  // LU decomposition of A
  if(sol.IsDone()) {          // is it OK ?
    sol.Solve(b1, x1);        // yes, so compute x1
    sol.Solve(b2, x2);        // then x2
    ...
  }
  else {                      // it is not OK:
    // fix up
    sol.Solve(b1, x1);        // error:
    // StdFail_NotDone is raised
  }
}
```

The next example demonstrates the use of the BissecNewton class, which implements a combination of the Newton and Bissection algorithms to find the root of a function known to lie between two bounds.The definition is an extract from the header file of the class math_BissecNewton:

Example

```
class BissecNewton {
   public:
      BissecNewton (math_FunctionWithDerivative& f,
                    const Standard_Real bound1,
                    const Standard_Real bound2,
                    const Standard_Real tolx);
      Standard_Boolean IsDone() const;
      Standard_Real Root();
};
```

The abstract class math_FunctionWithDerivative describes the services which have to be implemented for the function f which is to be used by a BissecNewton algorithm. The following definition corresponds to the header file of the abstract class math_FunctionWithDerivative:

Example

```
class math_FunctionWithDerivative {
   public:
      virtual Standard_Boolean Value
            (const Standard_Real x, Standard_Real& f) = 0;
      virtual Standard_Boolean Derivative
            (const Standard_Real x, Standard_Real& d) = 0;
      virtual Standard_Boolean Values
            (const Standard_Real x,
            Standard_Real& f,
            Standard_Real& d) = 0;
};
```

Now the test sample uses the BissecNewton class to find the root of the equation f(x)=x**2-4 in the interval [1.5, 2.5]:the function to solve is implemented in the class myFunction which inherits from the class math_FunctionWithDerivative,then the main program finds the required root.

Example

```
#include <math_BissecNewton.hxx>
#include <math_FunctionWithDerivative.hxx>
class myFunction : public math_FunctionWithDerivative
{
  Standard_Real coefa, coefb, coefc;

  public:
    myFunction (const Standard_Real a, const Standard_Real b,
                const Standard_Real c) :
      coefa(a), coefb(b), coefc(c)
    {}

    virtual Standard_Boolean Value (const Standard_Real x,
                                    Standard_Real& f)
    {
```

```
      f = coefa * x * x + coefb * x + coefc;
    }

    virtual Standard_Boolean Derivative (const Standard_Real x,
                                         Standard_Real& d)
    {
      d = coefa * x * 2.0 + coefb;
    }

    virtual Standard_Boolean Values (const Standard_Real x,
                                     Standard_Real& f, Standard_Real& d)
    {
      f = coefa * x * x + coefb * x + coefc;
      d = coefa * x * 2.0 + coefb;
    }
};

main()
{
  myFunction f(1.0, 0.0, 4.0);
  math_BissecNewton sol(F, 1.5, 2.5, 0.000001);
  if(Sol.IsDone()) { // is it OK ?
    Standard_Real x = sol.Root(); // yes.
  }
  else { // no
    // here some code is needed to try another
    // method or to output an error message.
  }
. . .
```

4.7. Precision

On the OCCT platform, each object stored in the database should carry its own precision value. This is important when dealing with systems where objects are imported from other systems as well as with various associated precision values.

The *Precision* package addresses the daily problem of the geometric algorithm developer: what precision setting to use to compare two numbers. Real number equivalence is clearly a poor choice. The difference between the numbers should be compared to a given precision setting.

Do not write:

```
if (X1 == X2)
```

but instead write:

```
if (Abs(X1-X2) < Precision)
```

Also, to order real numbers, keep in mind that:

```
if (X1 < X2 - Precision)
```

is incorrect.

```
if (X2 - X1 > Precision)
```

is far better when *X1* and *X2* are high numbers.

This package proposes a set of methods providing precision settings for the most commonly encountered situations.

In Open CASCADE Technology, precision is usually not implicit; low-level geometric algorithms accept precision settings as arguments. Usually these should not refer directly to this package.

High-level modeling algorithms have to provide a precision setting to the low level geometric algorithms they call. One way is to use the settings provided by this package. The high-level modeling algorithms can also have their own strategy for managing precision. As an example the Topology Data Structure stores precision values which are later used by algorithms. When a new topology is created, it takes the stored value.

Different precision settings offered by this package cover the most common needs of geometric algorithms such as *Intersection* and *Approximation*.

The choice of a precision value depends both on the algorithm and on the geometric space. The geometric space may be either:

- a "real" space, 3d or 2d where the lengths are measured in meters, micron, inches, etc.
- a "parametric" space, 1d on a curve or 2d on a surface where numbers have no dimension.

The choice of precision value for parametric space depends not only on the accuracy of the machine, but also on the dimensions of the curve or the surface.

This is because it is desirable to link parametric precision and real precision. If you are on a curve defined by the equation *P(t)*, you would want to have equivalence between the following:

```
Abs(t1-t2) < ParametricPrecision
Distance (P(t1),P(t2)) < RealPrecision.
```

4.7.1. The Precision package

The Precision package offers a number of package methods and default precisions for use in dealing with angles, distances, intersections, approximations, and parametric space.

It provides values to use in comparisons to test for real number equalities.

- Angular precision compares angles.
- Confusion precision compares distances.
- Intersection precision is used by intersection algorithms.
- Approximation precision is used by approximation algorithms.

- Parametric precision gets a parametric space precision from a 3D precision.
- *Infinite* returns a high number that can be considered to be infinite. Use -*Infinite* for a high negative number. (See also: *IsNegativeInfinite*, *IsPositiveInfinite*.)

4.7.2. Standard Precision values

This package provides a set of real space precision values for algorithms. The real space precisions are designed for precision to *0.1* nanometers. The only unit available is the millimeter.

The parametric precisions are derived from the real precisons by the *Parametric* function. This applies a scaling factor which is the length of a tangent to the curve or the surface. You, the user, provide this length. There is a default value for a curve with *[0,1]* parameter space and a length less than 100 meters.

The geometric packages provide Parametric precisions for the different types of curves.

The Precision package provides methods to test whether a real number can be considered to be infinite.

Angular returns Real from Standard;

Used to compare two angles. Current value is *Epsilon(2 * PI)* i.e. the smallest number *x* such that *2*PI + x* is different of *2* PI*.

Example

```
Confusion of two angles
Abs(Angle1 - Angle2) < Precision::Angular()

Parallelism of two vectors (Vec from gp)
V1.IsParallel(V2,Precision::Angular())
```

Note that *Precision::Angular()* can be used on both dot and cross products because for small angles the *Sine* and the *Angle* are equivalent. So to test if two directions of type *gp_Dir* are perpendicular, it is legal to use the following code:

```
Abs(D1 * D2) < Precision::Angular()
```

Confusion returns Real from Standard;

Used to test 3d distances. The current value is *1.e-7* in other words, 1/10 micron if the unit used is the millimeter.

Example

```
Confusion of two points (Pnt from gp)
P1.IsEqual(P2,Precision::Confusion())

A vector of null length (Vec from gp)
V.Magnitude() < Precision::Confusion()
```

Intersection returns Real from Standard;

A reasonable precision to pass to an Intersection process as a limit of refinement of Intersection Points. *Intersection* is high enough for the process to converge quickly. *Intersection* is lower than *Confusion* so that you still get a point on the intersected geometries. The current value is *Confusion() / 100*.

Approximation returns Real from Standard;

A reasonable precision to pass to an approximation process as a limit of refinement of fitting. Approximation is greater than the other precisions because it is designed to be used when time is at a premium. It has been provided as a reasonable compromise by the designers of the Approximation algorithm. The current value is *Confusion() * 10*.

Note that Approximation is greater than Confusion, so care must be taken when using Confusion in an approximation process.

PConfusion(T : Real from Standard) returns Real from Standard;

Used to test distances in parametric space.

This is *Precision::Parametric(Precision::Confusion(),T)*.

PIntersection(T : Real from Standard) returns Real from Standard;

Used for Intersections in parametric space.

This is *Precision::Parametric(Precision::Intersection(),T)*.

PApproximation(T : Real from Standard) returns Real from Standard;

Used for Approximations in parametric space.

This is *Precision::Parametric(Precision::Approximation(),T)* .

PConfusion returns Real from Standard;

Used to test distances in parametric space on a default curve.

This is *Precision::Parametric(Precision::Confusion())* .

PIntersection returns Real from Standard;

Used for Intersections in parametric space on a default curve.

This is *Precision::Parametric(Precision::Intersection())* .

PApproximation returns Real from Standard;

Used for Approximations in parametric space on a default curve.

This is *Precision::Parametric(Precision::Approximation())*

IsPositiveInfinite(R : Real from Standard) returns Boolean;

Returns True if *R* may be considered as a positive infinite number. The criterion is *R >1.e100*.

IsNegativeInfinite(R : Real from Standard) returns Boolean;

Returns True if *R* may be considered as a negative infinite number. The criterion is *R <-1.e100*.

5. Data Storage

5.1. Saving and Opening Files

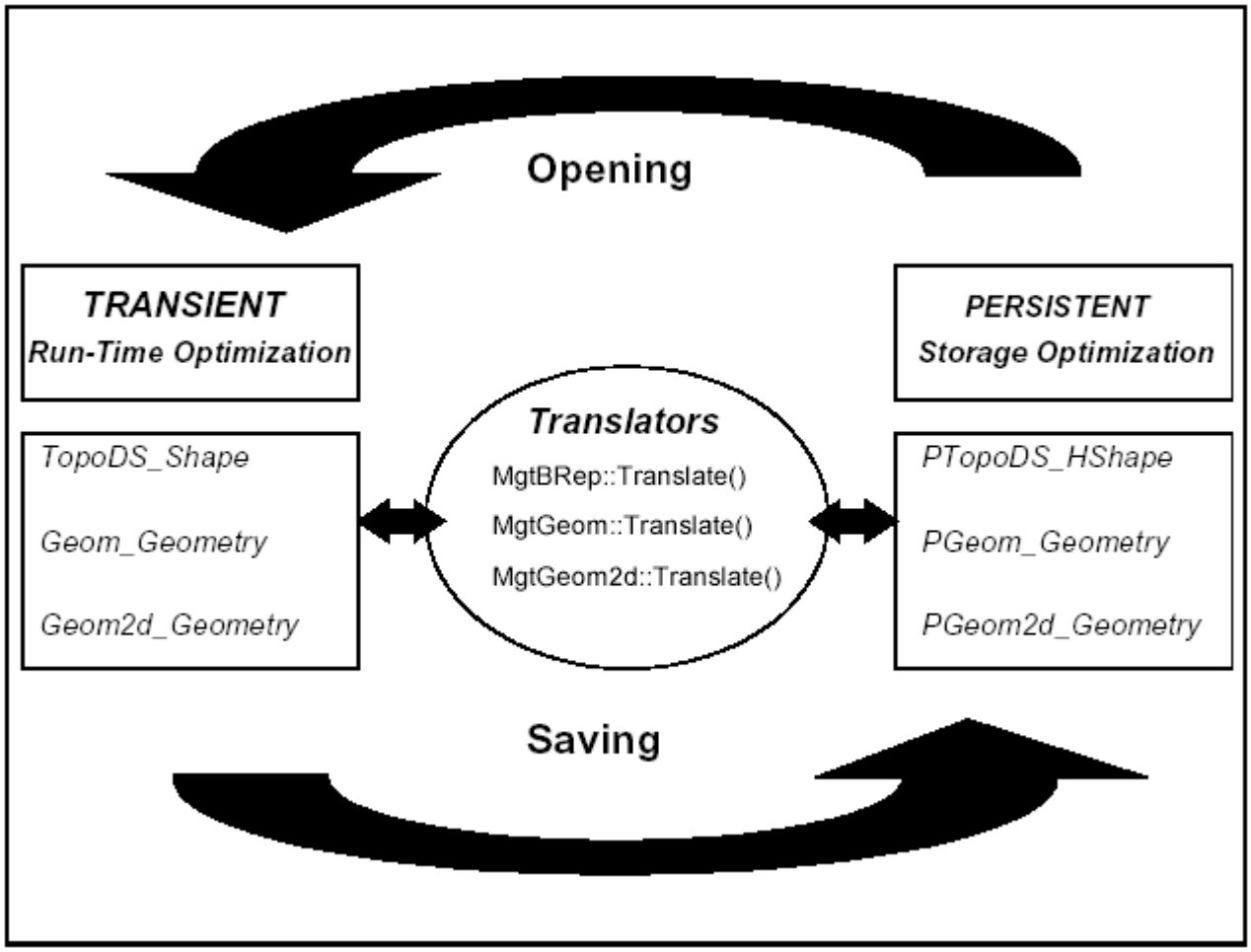

Figure 8. Saving-Opening mechanism overview

Roots of the transferable transient objects (in the example above, TopoDS_Shape, Geom_Geometry and Geom2d_Geometry) are used in algorithms; they contain data and temporary results.

The associated objects in the persistent domain are here PTopoDS_HShape, PGeom_Geometry and PGeom2d_Geometry. They contain a real data structure which is stored in a file.

Note that when an object is stored, if it contains another stored object, the references to the contained object are also managed.

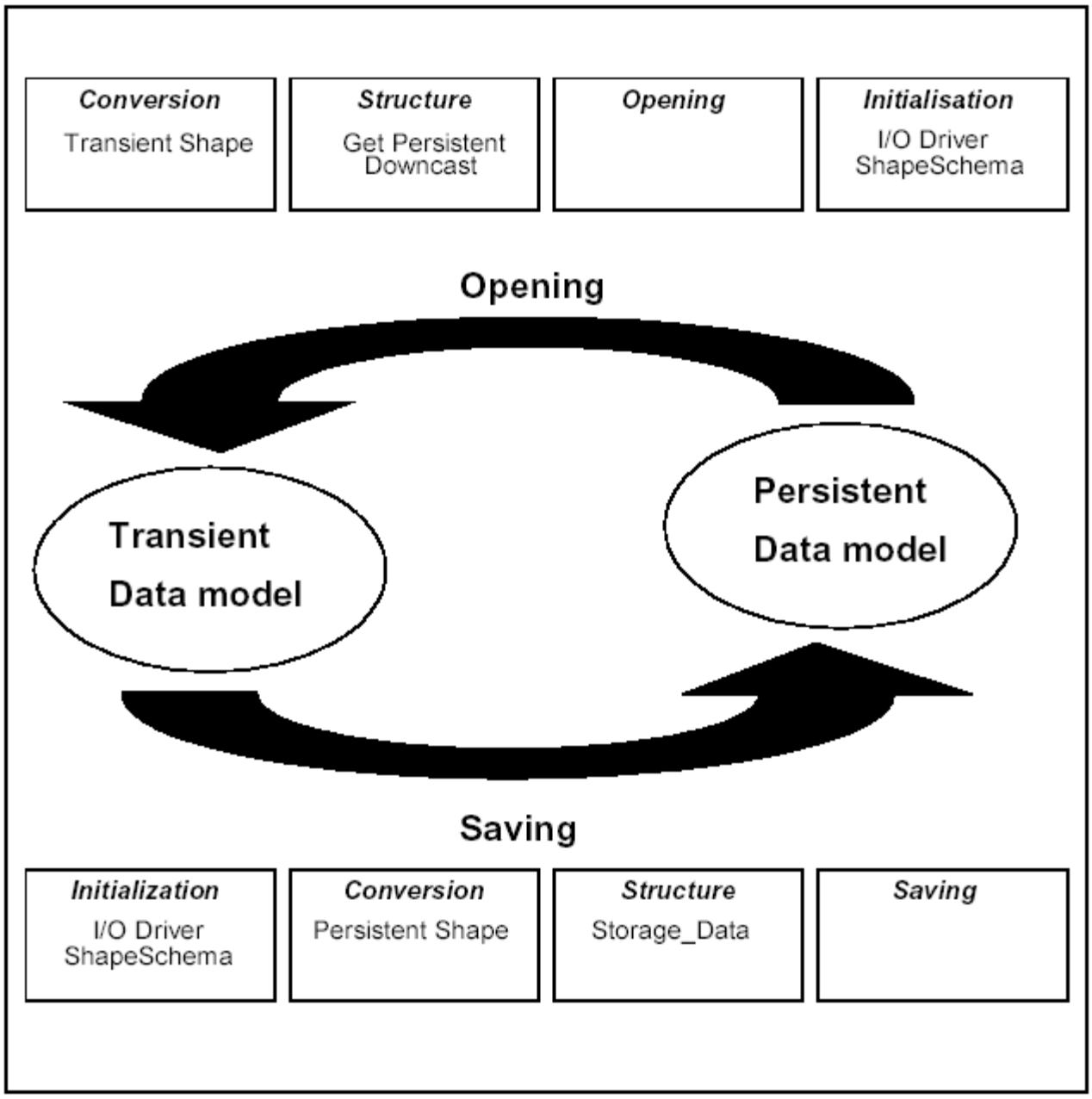

Figure 2. Saving-Opening mechanism

5.2. Basic Storage Procedures

The following illustrates the storage and retrieval mechanisms on shapes.

5.2.1. Saving

From a transient object, the storage procedure follows five main steps.

1. Create an I/O driver for files. For example:
   FSD_File f();

2. Instance the data schema which will process your persistent information. The schema is used for read/write operations. If ShapeSchema is the name of your schema:

```
Handle(ShapeSchema) s = new ShapeSchema;
```

3. Create the persistent shape from the transient shape.

```
TopoDS_Shape aShape;

PTColStd_TransientPersistentMap aMap;

Handle(PTopoDS_HShape) aPShape = MgtBRep::Translate
                                        (aShape, aMap, MgtBRep_WithoutTriangle);
```

4. Create a new container and fill it using the AddRoot() method

```
Handle(Storage_Data) d = new Storage_Data;

d -> AddRoot ("ObjectName", aPShape);
```

You may add as many objects as you want in this container.

5. Save to the archive

```
s -> Write (f,d);
```

5.2.2. Opening

The retrieval mechanism is the opposite of the storage mechanism. The procedure for retrieving an object is the following:

1. Create an I/O driver and instance a data schema (if not done)
2. Read the persistent object from the archive and get the list of objects using the Roots() method.

```
Handle(Storage_Data) d = s -> Read(f);

Handle(Storage_HSeqOfRoot) roots = d-> Roots();
```

3. Loop on root objects to get Standard_Persistent objects (the following sequence only gets the first root):

```
Handle(Standard_Persistent) p;

Handle(Standard_Root) r;

if(roots -> Length() >= 1) {

 r = roots -> Value(1);

 p = r -> Object();

}
```

4. DownCast the persistent object to a PTopoDS_Hshape

```
Handle(PTopoDS_HShape) aPShape;

aPShape = Handle(PTopoDS_HShape)::DownCast(p);
```

5. Create the TopoDS_Shape

```
TopoDS_Shape aShape;

PTColStd_PersistentTransientMap aMap;
```

MgtBRep::Translate (aPShape, aMap, aShape, MgtBRep_WithoutTriangle);

5.3. Methods Used

5.3.1. Write

This method is used to save data in a file. Its syntax is:

void Write(Storage_BaseDriver& s,

const Handle(Storage_Data)& aData);

5.3.2. Read

This method is used to open a storage file. Its syntax is:
Handle_Storage_Data Read(Storage_BaseDriver& s) const;