



Open CASCADE Technology  
6.7.1

Foundation Classes

April 29, 2014

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Foundation Classes Overview	1
1.2	Fundamental Concepts	3
1.2.1	Modules and toolkits	3
1.2.2	Packages	3
1.2.3	Classes	4
1.2.4	Genericity	5
1.2.5	Inheritance	6
1.2.6	Categories of Data Types	6
1.2.7	Exceptions	7
1.2.8	Persistence and Data Schema	7
<b>2</b>	<b>Basics</b>	<b>8</b>
2.1	Data Types	8
2.1.1	Primitive Types	8
2.1.2	Types manipulated by value	9
2.1.3	Types manipulated by reference (handle)	10
2.1.4	Summary of properties	10
2.2	Programming with Handles	10
2.2.1	Handle Definition	10
2.2.2	Type Management	11
2.2.3	Using Handles to Create Objects	12
2.2.4	Invoking Methods	13
2.2.5	Handle deallocation	13
2.2.6	Creating Transient Classes without CDL	14
2.3	Memory Management in Open CASCADE Technology	14
2.3.1	Usage	15
2.3.2	Configuring the memory manager	15
2.3.3	Implementation details	15
2.4	Exception Handling	16
2.4.1	Raising an Exception	17
2.4.2	Handling an Exception	18
2.4.3	Implementation details	19

# 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/](http://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

Root classes 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 *ExprIntpr*)
- 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.

## 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.

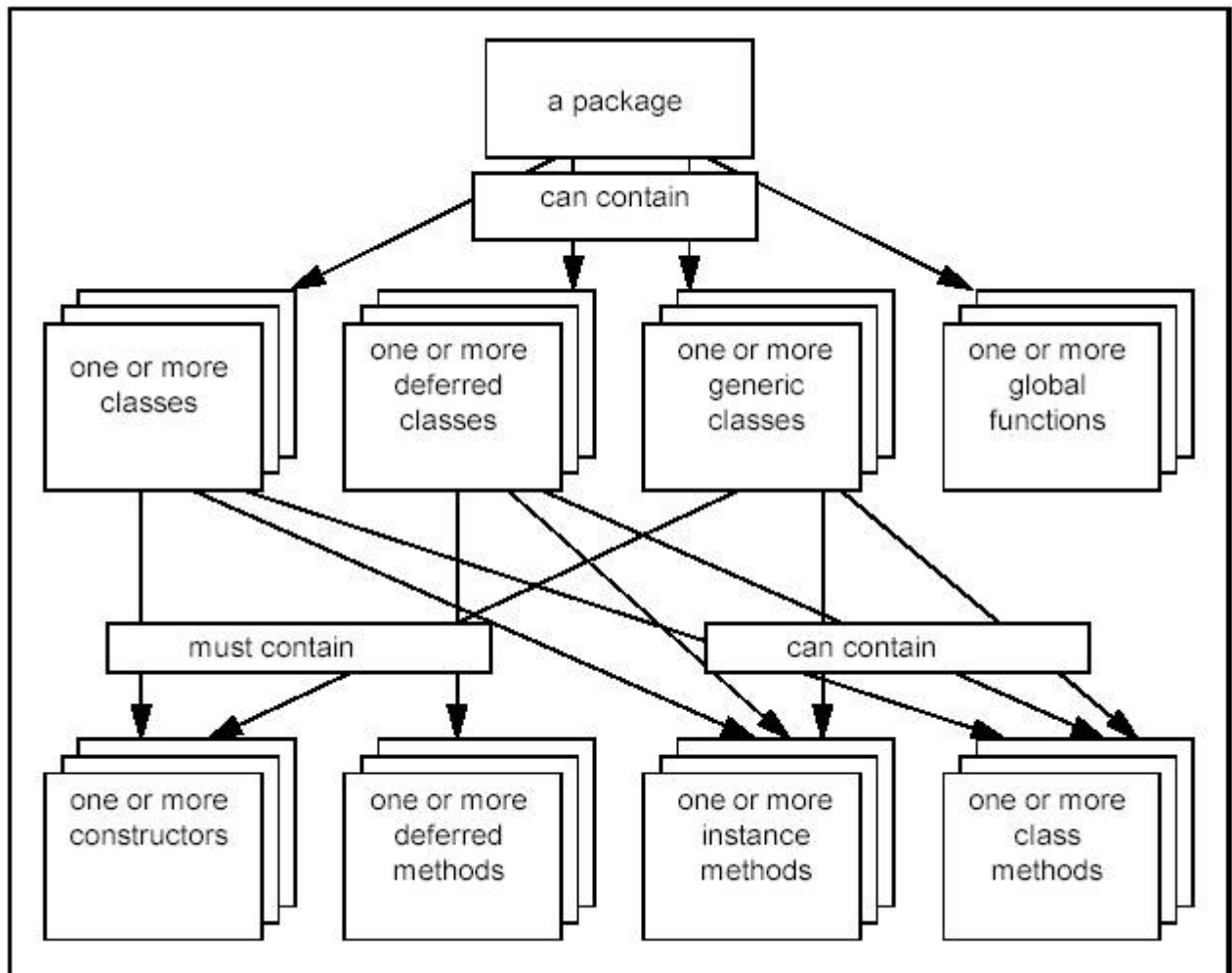


Figure 1: Contents of a package

**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. 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 descendants. 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.
- Generic classes. 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.

```
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'.

```
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

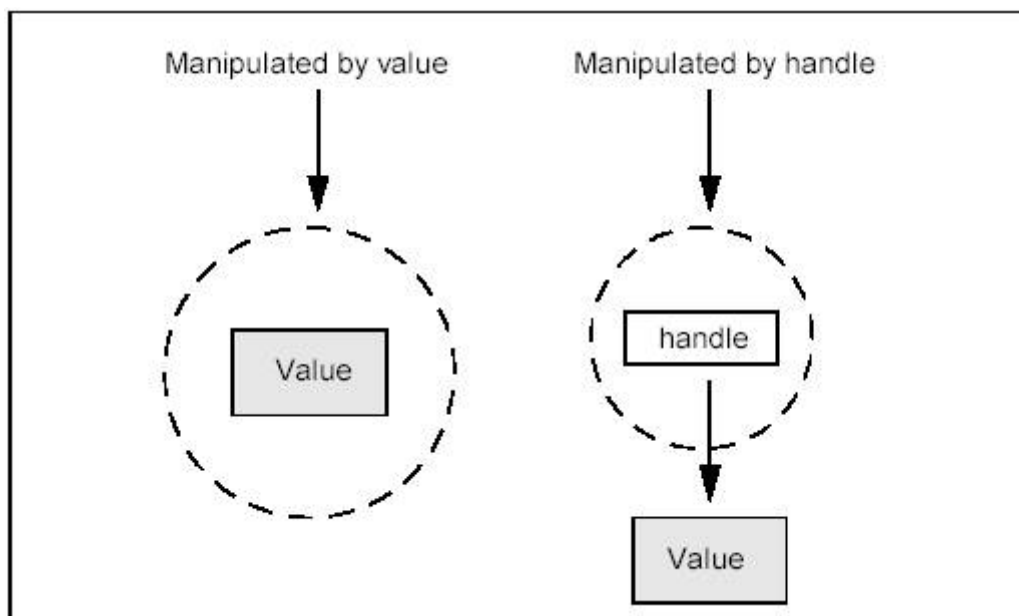


Figure 2: Manipulation of data types

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++:



```
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** is a 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 equivalence 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
char	Standard_Character
short	Standard_ExtCharacter
char*	Standard_CString
void*	Standard_Address
short*	Standard_ExtString

- The types with asterisk are pointers.

#### 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** : 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* and *not* as well as equivalence relations  $==$  and  $!=$  are defined on Booleans.
- **Standard\_Character** : 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** : 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** : 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** : fundamental type representing a generic pointer. *Address* is implemented as a *typedef* of the C++ *void* fundamental type.
- **Standard\_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.

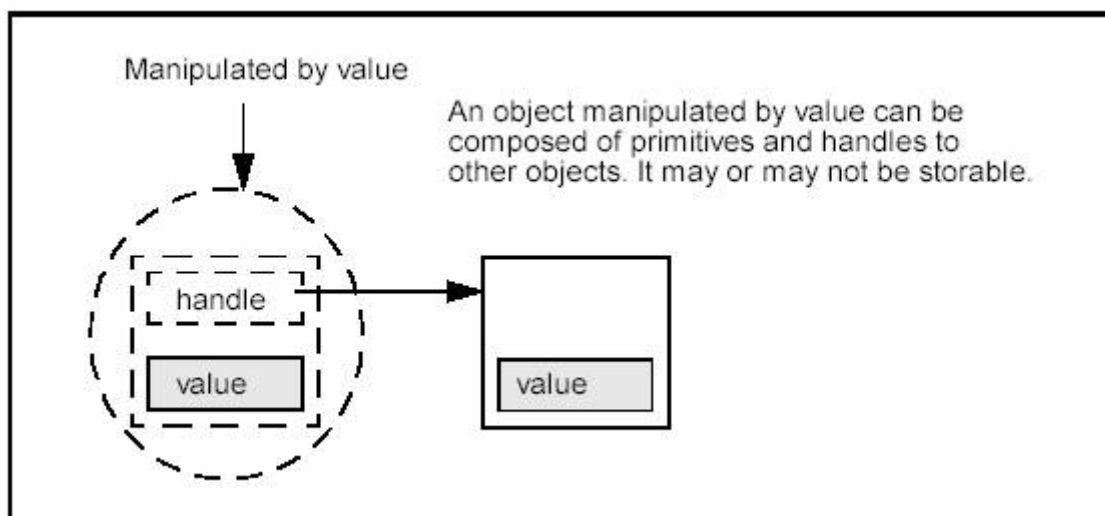


Figure 3: Manipulation of a data type by value

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.

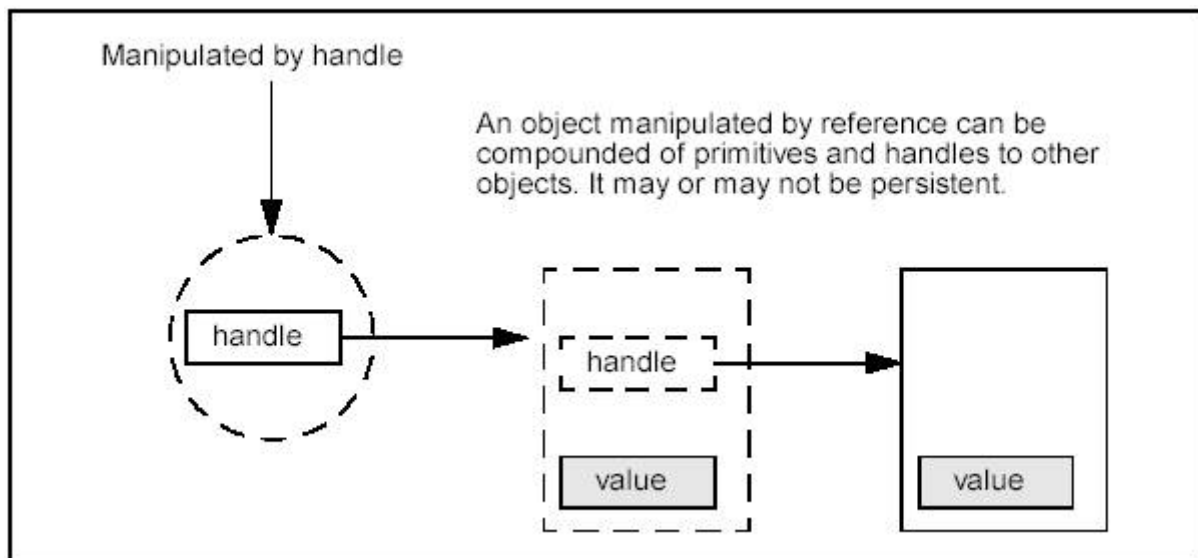


Figure 4: Manipulation of a data type by reference

### 2.1.4 Summary of properties

The following table summarizes how various data types are handled and stored.

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

## 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 preprocessor 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:

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

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** that 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:

```
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.

```
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:

```
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 transient objects *SequenceOfTransient* and two classes A and B that both inherit from *Standard\_Transient*, you get the following syntax:

```
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 instantiated.

```
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:

```
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:

```
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.

For example, we can find the maximum degree of a Bezier curve:

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

### 2.2.5 Handle deallocation

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 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.

```
...
{
    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)** - 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)** - 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).

In *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)
```

In *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 deallocating 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** that 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 for *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\_CELL\_SIZE*: 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** it is recommended to use options *MMGT\_OPT=2* and *MMGT\_REENTRANT=1* for applications that use OCCT memory manager from more than one thread, on multiprocessor hardware.

### 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\_CELL\_SIZE*, 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\_CELL\_SIZE* 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.

```
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 a *DomainError* type as follows:

```
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:

- *Value* function to extract an element
- *Lower* function to extract the lower bound of the array
- *Upper* function to extract the upper bound of the array.

then, the *Value* function may be implemented as follows:

```
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:

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

Using this syntax, the `Value` function becomes:

```
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 parent type of the three others.

```
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.

```

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:

```

#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 << endl;
    }
    return 1;
}

```

In this example 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).

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* needs to 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 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