VisualAge® C++ Professional for AIX®

IBM® Open Class™: Collections

Version 5.0

Note!
Before using this information and the product it supports, be sure to read the general information under "Notices" on page v.

Edition Notice

This edition applies to Version 5.0 of IBM VisualAge C++ and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Contents

Notices

Note to U.S. Government Users Restricted Rights -- use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd.
1150 Eglinton Avenue East
Toronto, Ontario M3C 1H7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

® (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. ® Copyright IBM Corp. 1998, 2000. All rights reserved.

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interface allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States, or other countries, or both:

AIX
AS/400
DB2
CICS
C Set ++
IBM
Network Station
Object Connection
OS/2
OS/390
OS/400
Open Class
Operating System/2
Operating System/400
PowerPC 403
PowerPC 601
PowerPC 603
PowerPC 604
Presentation Manager
RS/6000
S/390
SAA
Systems Application Architechture
TeamConnection
VisualAge
WebSphere
Workplace Shell

Lotus, Lotus Notes, and Domino are trademarks or registered trademarks of the Lotus Development Corporation in the United States, or other countries, or both.

Tivoli Management Environment, TME 10, and Tivoli Module Designer are trademarks of Tivoli Systems Inc. in the United States, or other countries, or both.

Encina and DCE Encina Lightweight Client are trademarks of Transarc Corporation in the United States, or other countries, or both.

Microsoft, Win32, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

UNIX is a registered trademark in the U.S. and other countries licensed exclusively through X/Open Company Limited.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks or registered trademarks of Intel Corporation in the United States, or other countries, or both.

C-bus is a registered trademark of Corollary, Inc.

PC Direct is a registered tradmark of Ziff Communicatoins Company and is used by IBM Corporation under license

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.

Industry Standards

VisualAge C++ Professional for AIX, Version 5.0 supports the following standards:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899–1990 [1992]). This standard has officially replaced American National standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. VisualAge C++ supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The IBM Systems Application Architecture (SAA) C Level 2 language definition.
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The ISO/IEC 9945–1:1990/IEEE POSIX 1003.-1990 standard.
- The X/Open Common Applications Environment Specifications, System Interfaces and Headers, Issue 4.

About This Book

The information in this PDF document is also available in the online help.

To find this information, or any topics listed in this document as Related Concepts, Related Tasks, or Related References, simply type the topic title into the search bar in the top frame of your browser in the online help.

For some topics, the suggested references may already be contained in this document. In such cases, there is a cross-reference to the page on which the related topic appears.

Chapter 1. Collection Classes Overview

Collections are used to store and manage elements (or objects) of a user-defined type. Different collections have different internal structures, performance characteristics, and access methods for storage and retrieval of objects. The Collection Classes implement the common collection types, such as trees, relations and ordered lists. They provide a framework of properties to help you decide which abstract collection type is appropriate in a given situation, and allow you to choose how the abstract data type you have chosen is implemented.

The Collection Classes let you choose the appropriate collection type for a given situation by providing collection classes which have a systematic and consistent combination of basic properties. These properties help you to select the appropriate level of abstraction. For example, you may have the choice between using a bag and a key sorted set. The properties of these two collections will help you decide which one is more appropriate.

The Collection Classes offer you a choice of implementations for each type of collection. Each abstract collection class has a common interface with all of its possible implementations. It is easy to replace one implementation with another for performance reasons or if the requirements of your application change.

Types of Classes
The Collection Classes are divided into three types:

Type	Description
Trees	Recursive collections of nodes, where each node holds an element and has a given number of nodes as children.
Flat collections	The most common types of collections, include abstractions such as sequence, set, bag, and map. Unlike trees, flat collections have no hierarchy of elements or recursive structure.
Auxiliary classes	Support other classes and include classes for cursors, iterators, and simple and managed pointers.

The Standard Template Library and the Collection Classes
We recommend that you use the Standard Template Library (STL) instead of the IBM Open Class Collections. The topic Migrating from IBM Collections to ANSI STL outlines the limitations and peculiarities of the STL, and serves as a guide for substituting STL containers with IBM Collections.

RELATED CONCEPTS

Hierarchy and Design of the Collection Classes
"Collection Characteristics" on page 2
Types of Collections

RELATED TASKS

"Chapter 2. Instantiate the Collection Classes" on page 25

RELATED REFERENCES

Introduction to the Standard Template Library
Migrating from IBM Collections to ANSI STL

Collection Characteristics

Four basic properties are used to differentiate between different flat collections:

Property	Description
Ordering	Whether a next or previous relationship exists between elements.
Access by key	Whether a part of the element (a key) is relevant for accessing an element in the collection. When keys are used, they are compared using relational operators.
Equality for elements	Whether equality is defined for the element.
Uniqueness of entries	Whether any given element or key is unique, or whether multiple occurrences of the same element or key are allowed.

Ordering of Collection Elements
The elements of a flat collection class can be ordered in three ways:

- Unordered collections have elements that are not ordered.
- Sorted collections have their elements sorted by an ordering relation defined for the element type. For example, integers can be sorted in ascending order, and strings can be ordered alphabetically. The ordering relation is determined by the instantiations for the collection class. For elements where the ordering relation returns the same position, elements are added in chronological order.
- Sequential collections have their ordering determined by an explicit qualifier to the add function, for example, addAtPosition.

A particular element in a sorted collection can be accessed quickly by using the ordering relation to determine its position. Unordered collections can also be implemented to allow fast access to the elements, by using, for example, a hash table or a sorted representation. The Collection Class Library provides a fast locate function that uses this structure for unordered and sorted collections. Even though unordered collections are often implemented by sorting the elements, do not assume that all unordered collections are implemented in this way. If your program requires this assumption to be true, use a sorted collection instead.

For each flat collection, the Collection Class Library provides both unordered and sorted abstractions. For example, the Collection Class Library supports both a set and a sorted set. The ordering property is independent of the other properties of flat collections. You can make a given flat collection unordered or sorted regardless of its other properties.

Access by Key
A given flat collection can have a key defined for its elements. A key is usually a data member of the element, but it can also be calculated from the data members of the element by some arbitrary function. Keys let you:

- Organize the elements in a collection

- Access a particular element in a collection

For collections that have a key defined, an equality relation must be defined for the key type. Thus, a collection with a key is said to have key equality.

Equality Relation
The Collection Class Library provides sorted and unsorted versions of maps and relations, for which both key and element equality must be defined. These collections are similar to key set and key bag, except that they define functions based on element equality, namely union and intersection. The add function behaves differently for maps and relations than it does for key set and key bag collections.

A flat collection can have an equality relation defined for its elements. The default equality relation is based on the element as a whole, not just on one or more of its data members (for example, the key). For two elements to be equal, all data members of both elements must be equal. The equality relation is needed for functions such as those that locate or remove a given element. A flat collection that has an equality relation has element equality.

The ISet and ISortedSet collections check element uniqueness using the less-than operator (operator<). However, you have to define the less-than operator so that it compares all the major parts of an element according to the requirements of your application. For example, suppose that you have an employee database. Each element in this database is a structure that contains three fields: employee name, employee serial number, and employee status report. To differentiate elements in this database you would compare the fields employee name and employee serial number, but you would not compare the employee status report field. Therefore for this database you would define a less-than operator that would compare only the employee name and the employee serial number fields.

Uniqueness of Entries
The terms unique and multiple relate to the key, in the case of collections with a key. For collections with no key, unique and multiple relate to the element.

In some flat collections, such as map, key set, and set, no two elements are equal or have equal keys. Such collections are called unique collections. Other collections, including relation, key bag, bag, and heap, can have two equal elements or elements with equal keys. Such collections are called multiple collections.

For those multiple collections that have keys with element equality (relation and sorted relation), elements are always unique while keys can occur multiple times. In other words, if element equality is defined for a multiple collection with key, element equality is tested before inserting a new element.

A unique collection with no keys and no element equality is not provided because a containment function cannot be defined for such a collection. A containment function determines whether a collection contains a given element.

The behavior during element insertion (when one of the add... methods is applied to a collection) distinguishes unique and multiple collections. In unique collections, the add function does not add an element that is equal to an element that is already in the collection. In multiple collections, the add function adds elements regardless of whether they are equal to any existing elements or not.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes
Types of Collections
"Examples of Using the Collection Classes" on page 6

RELATED TASKS

"Chapter 2. Instantiate the Collection Classes" on page 25

Types of Collections

There are two types of collections, categorized by their internal structures: *flat collections* and *trees*.

Flat Collections

Flat collections have no hierarchy of elements or recursive structure. A flat collection is like an array with added functionality. The following flat collections are provided by the Collection Class Library:

- sets: key sets, sorted sets, key sorted sets
- bags: key bags, sorted bags, key sorted bags
- queues: priority queues
- deques
- heaps
- maps: sorted maps, relations, sorted relations
- sequences and equality sequences
- stacks

Combinations of Properties for Flat Collections

The figure below shows the flat collection that results from each combination of properties. For example, Map appears in the Unique, Unordered column for the Key, Element Equality row. This means that a map is unordered, each element is unique, keys are defined, and element equality is defined. This implies that there are no flat collections that have all of the following properties:

- The collection is ordered
- The collection is sequential
- The collection allows an element to appear more than once
- Keys are defined for elements in the collection

The rationale for not implementing collections with these combinations of properties is that there is no reason to choose them over another collection that is already available. For example, for an ordered collection that is sequential and offers access by key, the key access would only have advantages if the elements are stored in a position depending on their key. Because they are not, there is no flat collection with key access that maintains a sequential order.

	Unordered		**Ordered**		
			Sorted		**Sequential**
	Unique	**Multiple**	**Unique**	**Multiple**	**Multiple**
• **Key*** • **Element Equality**	Map	Relation	Sorted map	Sorted relation	N/A

• **Key*** • **No Element Equality**	Key set	Key bag	Key sorted set	Key sorted bag	N/A
• **No Key** • **Element Equality**	Set	Bag	Sorted set	Sorted bag	Equality sequence
• **No Key** • **No Element Equality**	N/A	Heap	N/A	N/A	Sequence

*Key equality must be defined.

For example, the default set collection:

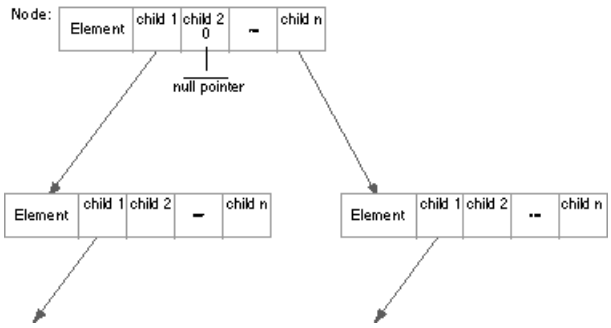
- does not allow an item to be added if it is already present in the collection, ensuring that elements are unique
- is not sorted
- cannot locate elements by key

Trees

Trees can be described either as structures whose elements have a hierarchy or as a special form of recursive structure. Recursively a tree can be described as a node (parent) with pointers to other nodes (children). Every node has a fixed number of pointers, which are set to null at initialization time. Insertion of a new node involves setting a pointer in the parent so that it points to the inserted child. The figure below illustrates the structure of an n-ary tree.

One node is the entry point to the tree. This node is designated as the root. Nodes without any pointers to other nodes are called leaf nodes or terminal nodes.

The Structure of N-ary Trees



Similarly, you can obtain tree-like or recursive structures by implementing the array of children of a node as a flat collection of nodes. This will give you different functionality for the children, for example, the ability to locate a child with a given value.

Trees in general are more useful for searching elements than for adding and deleting elements. For this reason, they are often called search trees. The descriptions of AVL and B* trees explain why trees are well-suited for searching. Generally, you can locate and insert elements in collections implemented as trees faster than you can in collections implemented as lists. However, if you only want to iterate through elements in a collection, it is faster to iterate through the elements of a list.

RELATED CONCEPTS

Collection Classes Overview
Hierarchy and Design of the Collection Classes
Collection Characteristics
Examples of Using Collection Classes

RELATED TASKS

Choose One of the Provided Implementation Variants

Examples of Using the Collection Classes

Bag
An example of using a bag is a program for entering observations on species of plants and animals found in a river. Each time you spot a plant or animal in the river, you enter the name of the species into the collection. If you spot a species twice during an observation period, the species is added twice, because a bag supports multiple elements. You can locate the name of a species that you have observed, and you can determine the number of observations of that species, but you cannot sort the collection by species, because a bag is an unordered collection. If you want to sort the elements of a bag, use a sorted bag instead.

Sorted Bag
An example of using a sorted bag is a program for entering observations on the types of stones found in a riverbed. Each time you find a stone on the riverbed, you enter the stone's mineral type into the collection. You can enter the same mineral type for several stones, because a sorted bag supports multiple elements. You can search for stones of a particular mineral type, and you can determine the number of observations of stones of that type. You can also display the contents of the collection, sorted by mineral type, if you want a complete list of observations made to date.

Key Bag
An example of using a key bag is a program that manages the distribution of combination locks to members of a fitness club. The element key is the number that is printed on the back of each combination lock. Each element also has data members for the club member's name, member number, and so on. When you join the club, you are given one of the available combination locks, and your name, member number, and the number on the combination lock are entered into the collection. Because a given number on a combination lock may appear on several locks, the program allows the same lock number to be added to the collection multiple times. When you return a lock because you are leaving the club, the program finds the elements whose key matches your lock's serial number, and deletes the matching element that has your name associated with it.

Key Sorted Bag
An example of using a key sorted bag is a program that maintains a list of families, sorted by the number of family members in each family. The key is the

number of family members. You can add an element whose key is already in the collection (because two families can have the same number of members), and you can generate a list of families sorted by size. You cannot locate a family except by its key, because a key sorted bag does not support element equality.

Heap and Stack
You can compare using a heap collection to managing the scrap metal entering a scrap yard. Pieces of scrap are placed in the heap in an arbitrary location, and an element can be added multiple times (for example, the rear left fender from a particular kind of car). When a customer requests a certain amount of scrap, elements are removed from the heap in an arbitrary order until the required amount is reached. You cannot search for a specific piece of scrap except by examining each piece of scrap in the heap and manually comparing it to the piece you are looking for.

An example of using a stack is a program that keeps track of daily tasks that you have begun to work on but that have been interrupted. When you are working on a task and something else comes up that is more urgent, you enter a description of the interrupted task and where you stopped it into your program, and the task is pushed onto the stack. Whenever you complete a task, you ask the program for the most recently saved task that was interrupted. This task is popped off the stack, and you resume your work where you left off. When you attempt to pop an item off the stack and no item is available, you have completed all your tasks.

Map
An example of using a map is a program that translates integer values between the ranges of 0 and 20 to their written equivalents, from their written forms to their numeric forms. Two maps are created, one with the integer values as keys, one with the written equivalents as keys. You can enter a number, and that number is used as a key to locate the written equivalent. You can enter a written equivalent of a number, and that text is used as a key to locate the value. A given key always matches only one element. You cannot add an element with a key of 1 or “one” if that element is already present in the collection.

Sorted Map
An example of using a sorted map is a program that matches the names of rivers and lakes to their coordinates on a topographical map. The river or lake name is the key. You cannot add a lake or river to the collection if it is already present in the collection. You can display a list of all lakes and rivers, sorted by their names, and you can locate a given lake or river by its key, to determine its coordinates.

Queue
An example of using a queue is a program that processes requests for parts at the cash sales desk of a warehouse. A request for a part is added to the queue when the customer’s order is taken, and is removed from the queue when an order picker receives the order form for the part. Using a queue collection in such an application ensures that all orders for parts are processed on a first-come, first-served basis.

Deque
An example of using a deque is a program for managing a lettuce warehouse. Cases of lettuce arriving into the warehouse are registered at one end of the queue (the “fresh” end) by the receiving department. The shipping department reads the other end of the queue (the “old” end) to determine which case of lettuce to ship

next. However, if an order comes in for very fresh lettuce, which is sold at a premium, the shipping department reads the "fresh" end of the queue to select the freshest case of lettuce available.

Priority Queue
An example of a priority queue is a program used to assign priorities to service calls in a heating repair firm. When a customer calls with a problem, a record with that person's name and the seriousness of the situation is placed in a priority queue. When a service person becomes available, customers are chosen by the program beginning with those whose situation is most severe. In this example, a serious problem such as a nonfunctioning furnace would be indicated by a low value for the priority, and a minor problem such as a noisy radiator would be indicated by a high value for the priority.

Relation
An example of using a relation is a program that maintains a list of all your relatives, with an individual's relationship to you as the key. You can add an aunt, uncle, grandmother, daughter, father-in-law, and so on. You can add an aunt even if an aunt is already in the collection, because you can have several relatives who have the same relationship to you. (For unique relationships such as mother or father, your program would have to check the collection to make sure it did not already contain a family member with that key, before adding the family member.) You can locate a member of the family, but the family members are not in any particular order.

Sorted Relation
An example of using a sorted relation is a program used by telephone operators to provide directory assistance. The computerized directory is a sorted relation whose key is the name of the individual or business associated with a telephone number. When a caller requests the number of a given person or company, the operator enters the name of that person or company to access the phone number. The collection can have multiple identical keys, because two individuals or companies might have the same name. The collection is sorted alphabetically, because once a year it is used as the source material for a printed telephone directory.

Sorted Set
An example of using a sorted set is a program that tests numbers to see if they are prime. Two complementary sorted sets are used, one for prime numbers, and one for nonprime numbers. When you enter a number, the program first looks in the set of nonprime numbers. If the value is found there, the number is nonprime. If the value is not found there, the program looks in the set of prime numbers. If the value is found there, the number is prime. Otherwise the program determines whether the number is prime or nonprime, and places it in the appropriate sorted set. The program can also display a list of prime or nonprime numbers, beginning at the first prime or nonprime following a given value, because the numbers in a sorted set are sorted from smallest to largest.

Key Set
An example of using a key set is a program that allocates rooms to patrons checking into a hotel. The room number serves as the element's key, and the patron's name is a data member of the element. When you check in at the front desk, the clerk pulls a room key from the board, and enters that key's number and your name into the collection. When you return the key at check-out time, the record for that key is removed from the collection. You cannot add an element to the collection that is already present, because there is only one key for each room.

Key Sorted Set
An example of using a key sorted set is a program that keeps track of canceled credit card numbers and the individuals to whom they are issued. Each card number occurs only once, and the collection is sorted by card number. When a merchant enters a customer's card number into a point-of-sale terminal, the collection is checked to see if that card number is listed in the collection of canceled cards.

Sequence
An example of a sequence is a program that maintains a list of the words in a paragraph. The order of the words is obviously important, and you can add or remove words at a given position, but you cannot search for individual words except by iterating through the collection and comparing each word to the word you are searching for. You can add a word that is already present in the sequence, because a given word may be used more than once in a paragraph.

Equality Sequence
An example of using an equality sequence is a program that calculates, and places in a collection, members of the Fibonacci series, which is a series of integers in which each integer is equal to the sum of the two preceding integers. Multiple elements of the same value are allowed. For example, the sequence begins with two instances of the value 1. Element equality allows you to search for a given element, for example 8, and find out what element follows it in the sequence.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Types of Collections

RELATED TASKS

"Chapter 2. Instantiate the Collection Classes" on page 25

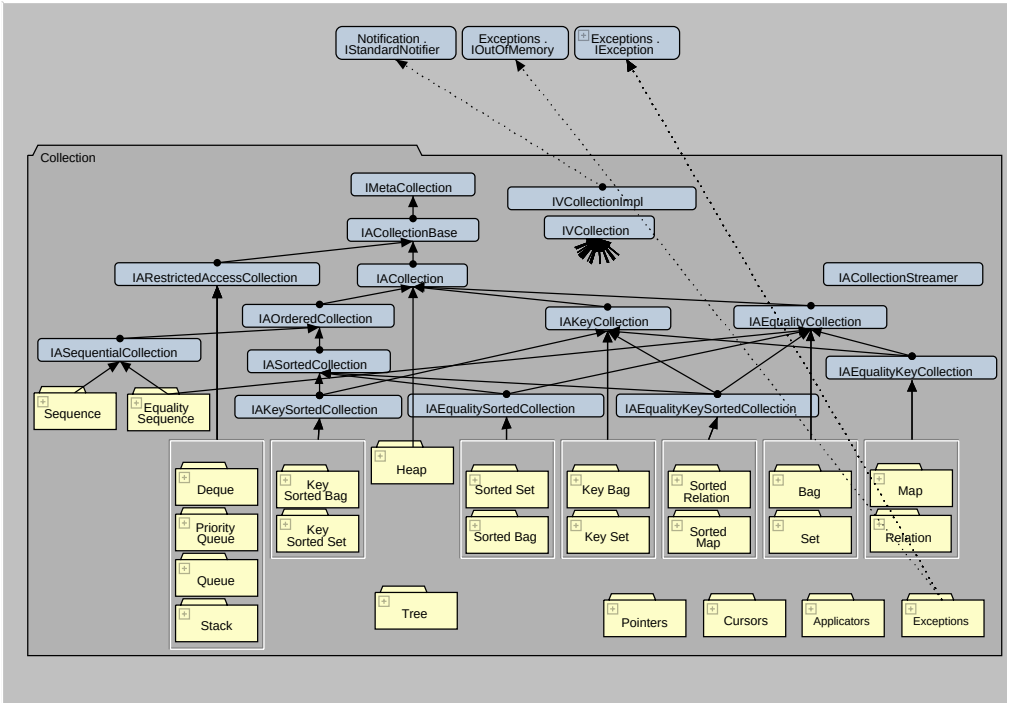
RELATED REFERENCES

IBag
ISortedBag
IKeyBag
IKeySortedBag
IHeap
IStack
IMap
ISortedMap
IQueue
IDeque
IPriorityQueue
IRelation
ISortedRelation
ISortedSet
IKeySet
IKeySortedSet
ISequence
IEqualitySequence

Hierarchy and Design of the Collection Classes

Collection Class Hierarchy

The classes in the Collection Classes are all related through the hierarchy of abstract classes shown in the figure below:



Overall Implementation Structure

The abstract collection classes represent a concept, and classes derived from it represent implementations of the concept. You cannot create any objects from the abstract classes. The names of the abstract collection classes start with the letters IA.

Three abstract collection classes inherit directly from IACollection, the abstract class that defines the base interfaces for the collection classes:

Abstract Class	Description
IAEqualityCollection	Defines the interfaces for the property of *element equality* (whether equality is defined for the element).
IAKeyCollection	Defines the interfaces for the *key property* (whether you use part of an element (a key) to access an element in the collection).
IAOrderedCollection	Defines the interfaces for the property of *ordered elements* (whether a next or previous relationship exists between elements).

Two abstract collection classes inherit from IAOrderedCollection:

Abstract Class	Description
IASortedCollection	Defines the interfaces for ordered collections that are sorted.
IASequentialCollection	Defines the interfaces for ordered collections that are not sorted.

More abstract collection classes are derived by inheriting from a combination of the above abstract classes. Concrete classes have the properties of the abstract classes that they inherit from:

Abstract Classes	Derived Abstract Class	Derived Concrete Classes
IAEqualityCollection		ISet IBag
IAKeyCollection		IKeyBag IKeySet
IASequentialCollection		ISequence
IAEqualityCollection IASequentialCollection		IEqualitySequence
IAEqualityCollection IAKeyCollection	IAEqualityKeyCollection	IMap IRelation
IAEqualityCollection IASortedCollection	IAEqualitySortedCollection	ISortedBag ISortedSet
IAKeyCollection IASortedCollection	IAKeySortedCollection	IKeySortedBag IKeySortedSet
IAEqualityCollection IAKeyCollection IASortedCollection	IAEqualityKeySortedCollection	ISortedMap ISortedRelation

The IARestrictedAccessCollection abstract class defines defines the interfaces for the restricted access collections. Restricted access collections let you use only a subset of functions of an existing collection. The IBM Open Class provides you with the following collections with restricted access:

- IStack, IDeque, and IQueue, which are all based on ISequence
- IPriorityQueue, which is based on IKeySortedBag

Each abstract collection type has several possible implementations. Some of these implementations are basic; that is, the collection class is implemented directly as a concrete class. These basic implementations include the following:

- AVL trees
- Hash tables
- Linked sequences
- Tabular sequences

Variant implementations of the same collection behave externally in the same way but may offer improved performance for a particular application, depending on the collection's characteristics. Sets can be implemented, for example, as AVL trees, lists, or hash tables.

Default implementations are provided for every collection. Two default classes are provided for each abstract data type: a class that is instantiated only with the

element type (and possibly the key type) and one that is instantiated by passing in element-specific functions. In many cases, you do not need to concern yourself with the choice of implementation. If you choose not to specify one, the Collection Classes will use a reasonable default implementation class.

For example, the class ISet uses an AVL tree as the default implementation. The other implementation variants are linked list and diluted table. The three implementation variants ISetAsAvlTree (a set implemented as an AVL tree), ISetAsList (a set implemented as a linked list), and ISetAsDilTable (a set implemented as a diluted table) are subclasses of IASet. If you do not want to deal with implementation variants, you can just use the default class ISet.

With abstract classes, you can program to a more generalized interface without knowing what specific collection type your code will operate on. Implementation details can be left for later. For example, when working with a set, you can write your program to use the interfaces of the abstract classes IASet or IAEqualityCollection, rather than the concrete classes ISet, IGSet, ISetAsBstTree, and so on.

The Based-On Concept
The Collection Classes achieve a high degree of implementation flexibility by basing several collection class implementations on other abstract classes, rather than by implementing them directly through a concrete implementation variant of the class. This design feature results in an implementation path rather than the selection of an implementation in a single step. The Collection Classes contain type definitions for the most common implementation paths.

The element functions that are needed by a particular implementation depend on all collection class templates that participate in the implementation. For example, while ISet requires at least element equality to be defined, an AVL tree implementation of this set also requires the element type to provide a comparison function. A hash table implementation also requires the element type to have a hash function. The required element functions for all predefined implementation variants are listed for individual collection types.

For a concrete implementation, such as a set based on a key-sorted set that is in turn based on a tabular sequence, these class templates are plugged together so that the elements only need to define the operations that are needed for the specific type of collection being used.

Element Functions and Key-Type Functions
The member functions of the Collection Classes call other functions to manipulate elements and keys. These functions are called element functions and key-type functions, respectively. Member functions of the Collection Classes may, for example, use the element's assignment or copy constructors for adding an element, or they may use the element's equality operator for locating an element in the collection.

Collection Class Polymorphism
Polymorphic use of collections differs from polymorphism of the element type. Polymorphic use of collections means that a function can specify an abstract collection type for its argument, for example IACollection, and then accept any concrete collection given as its actual argument. Element polymorphism means that you can use the collections with any elements that provide basic operations like assignment and equality. This section deals with the polymorphic use of collections rather than elements.

Each abstract class is defined by its functions and their behavior. The most abstract view of a collection is a container without any ordering or any specific element or key properties. Elements can be added to a collection, and a collection can be iterated over. A polymorphic function on collections that uses only properties of the most abstract view might be to print all elements.

Collections with more specialized element properties, such as equality or key equality, also provide functions for retrieving element occurrences by a given element or key value. Ordered collections provide the notion of a well-defined ordering of element occurrences, either by an element ordering relation or by explicit positioning of elements within a sequence. Ordered collections define operations for positional element access. Sorted collections provide no further functions, but define a more specific behavior, namely that the elements or their keys are sorted.

The properties represented by abstract collection classes are combined through multiple inheritance. For example, the abstract collection class IAEqualitySortedCollection, for example, combines the properties of element equality and element sorting, which implies being ordered. If a polymorphic function uses IAEqualitySortedCollection as its argument type, the argument will be sorted, and the function can use functions such as contains that are only defined for collections with element equality.

Default and Variant Implementations
Each collection provides a default implementation. You can easily replace the default implementation of a collection by an implementation variant of the same collection that behaves externally in the same way but may offer improved performance for your application.

RELATED CONCEPTS

Collection Classes Overview
Types of Collections
Class Template Naming Conventions

RELATED TASKS

Instantiate the Collection Classes
Implement Element- and Key-Type Functionality

Class Template Naming Conventions

All class templates begin with an uppercase I. The table below shows the naming conventions used to distinguish between different types of class templates, given a default class template of ISet:

Class name	Meaning of letters
ISet	Default class template.
ISetImpl	Typeless implementation class.
ICSetImpl	Typeless implementation class that implements additional checks.
IGSet	Default generic class template. The element operations class can be specified as template argument.
IASet	Abstract class template.
IVSet	Default notification-enabled class.

Class name	Meaning of letters
ISetAsAvlTree ISetAsBstTree ISetAsList ISetAsDilTable ISetAsHshTable ISetAsTable	Variant class templates.
IGSetAsBstTree IGSetAsList IGSetAsDilTable IGSetAsHshTable IGSetAsTable	Variant generic class templates.
IVSetAsAvlTree IVSetAsDilTable IVSetAsHshTable IVSetAsTable IVSetAsList IVSetAsBstTree	Variant notification-enabled classes.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Types of Collections
Possible Implementation Paths

RELATED TASKS

"Choose One of the Provided Implementation Variants" on page 23

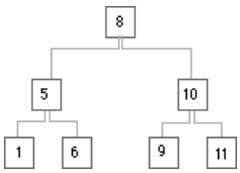
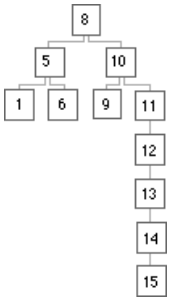
Implementation Variants

AVL Tree

AVL trees are a special form of binary tree. You can better understand AVL trees if you know how a binary tree is structured.

Trees are binary trees when all nodes have either zero, one, or two children. Binary trees are often used in applications where you want to store elements in a certain order. In such cases, the left child always points to an element that comes earlier in the order than the parent node, and the right child points to an element that comes later than the parent. A search through a binary tree begins at the root node. The search then continues downward until the desired element is found, by determining whether a node comes before or after the searched-for node, and then following the appropriate branch. For example, the binary tree shown in the figure below has elements added in the following sequence: 8 - 10 - 5 - 1 - 9 - 6 - 11. A search for element 9 begins at the root node (element 8). Assuming that the element value defines the ordering relation, the search would take the right node from element 8 (because 9 is greater than 8) and would arrive at element 10. The search would take the left node from element 10 (because 9 is smaller than 10) and would arrive at element 9, the desired element.

The following figures show you a binary search tree and an unbalanced binary search tree.

Balanced Binary Search Tree	Unbalanced Binary Search Tree
	

One drawback of a binary search tree is that the tree can easily become unbalanced. The figure shows how unbalanced the tree becomes when the elements 12 through 15 are added. The unbalanced binary tree looks almost like a list, without the performance advantage of a normal binary search tree. To obtain this performance advantage, a binary search tree should always remain balanced. The AVL Tree is a special form of binary search tree that maintains balance.

AVL trees are useful for collections containing a large number of small elements. An AVL tree implementation is even suitable for adding and deleting, because the performance overhead for the rebalancing that sometimes occurs when an element is added or deleted is still less expensive than searching through the elements of a sequence to find the position at which to add or delete an element.

If you use a set collection and do not choose an implementation variant, you are automatically using an AVL tree. If you use a set and are not aware that the set is implemented as an AVL tree, you may be surprised that a set requires an ordering relation, when a set is an unordered collection. The reason a set requires an ordering relation is that an AVL tree requires an ordering relation to determine where to add new elements or where to find elements to be accessed or deleted. As this example shows, required element and key-type functions are determined by two factors:

- Some functions are required because of the properties of the collection.
- Some properties are required because of the implementation variant you choose.

RELATED CONCEPTS

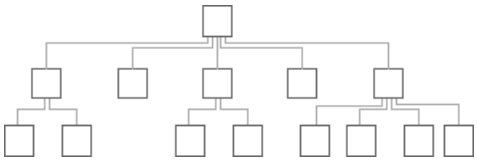
"Chapter 1. Collection Classes Overview" on page 1
Types of Collections
Possible Implementation Paths
"Class Template Naming Conventions" on page 13
"B* Tree" on page 16
"Diluted Table" on page 17
"Hash Table" on page 18
"List" on page 19
"Table" on page 20

RELATED TASKS

"Choose One of the Provided Implementation Variants" on page 23
"Chapter 2. Instantiate the Collection Classes" on page 25

B* Tree

A B* tree is a search tree that may have more than two references per node.The figure below shows a B* tree with up to five children per node:



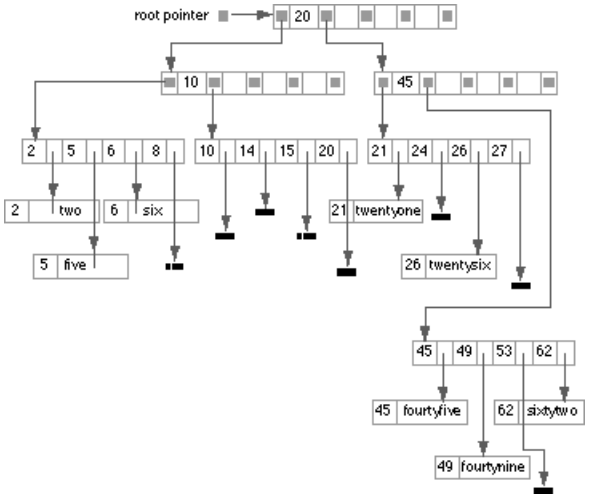
A B* tree combines the advantages of binary search and sequential access upon the same set of keys. B* trees are based on two simple ideas:

- The internal nodes are used only for storing the keys, with all real data stored at the leaves. A B* tree takes into consideration the page or block size of the operating system's virtual memory structure, and is suitable for applications where paging or memory thrashing is a constraint.
- The leaves of a B* tree are chained together in logical sequence to support sequential access.

A B* tree implementation variant is suitable when you have many large elements that are accessed by key. Because keys and their data are separated, the keys in the tree structure are used for a quick search and the pointers are used for quick access to the data.

In contrast to a B* tree, keys and data in an AVL tree are both stored in the nodes. This means that searching through elements could cause page faults if the elements are large, because the various keys may be spread across several pages along with the data they refer to.

In the figure below, the B* tree has an order of 5 (which means that each internal node has a maximum of five references):



The data is stored only in the leaves. A leaf block is built to hold one element. A leaf block may be larger than one page. The B* tree implementation uses the keys in the nodes for quick access to a required page (leaf), or it uses the keys for a quick sequential access to all pages, and hence to all elements.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Types of Collections
Possible Implementation Paths
"Class Template Naming Conventions" on page 13
"AVL Tree" on page 14
"Diluted Table"
"Hash Table" on page 18
"List" on page 19
"Table" on page 20

RELATED TASKS

"Choose One of the Provided Implementation Variants" on page 23
"Chapter 2. Instantiate the Collection Classes" on page 25

Diluted Table

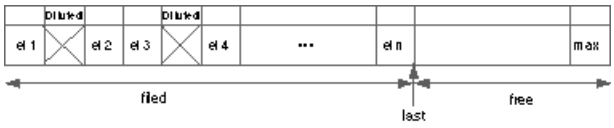
A diluted table, like a table sequence, is an array implementation of a list. However, when you delete an element from a diluted table, it is not actually deleted, but only flagged as deleted. This provides a performance advantage, in that elements following a deleted element do not need to be shifted. The additional overhead of using a dilution flag is trivial.

If you want to add a new element at a certain position, only those elements between that position and the next element flagged as deleted need to be shifted. (If no elements later in the list are flagged as deleted, then all elements beyond the insertion position must be shifted.)

Use a diluted table rather than a table if your application will be doing much adding or deleting of elements after the collection is established.

The figure below shows a diluted table implementation variant.

Diluted Table Implementation Variant



RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Types of Collections
Possible Implementation Paths
"Class Template Naming Conventions" on page 13
"AVL Tree" on page 14
"B* Tree" on page 16
"Hash Table" on page 18
"List" on page 19
"Table" on page 20

RELATED TASKS

"Choose One of the Provided Implementation Variants" on page 23
"Chapter 2. Instantiate the Collection Classes" on page 25

Hash Table

Hashing is another important and widely used technique for implementing collections. Conceptually, hashing involves calculating an index from the key or other parts of an element, and then using that index to look for matches in a hash table. The function that calculates the index is called a hash function.

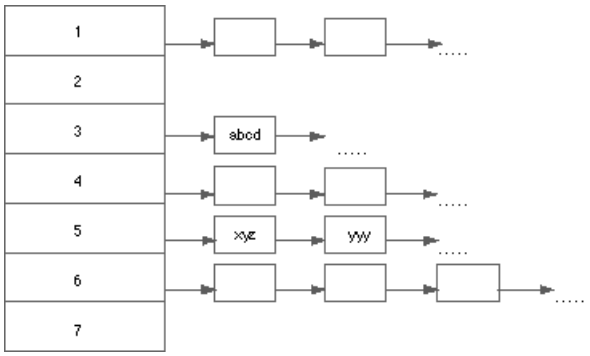
A hash-table implementation variant is suitable for nearly all applications with a balanced mix of operations. Such an implementation is quick for retrieving elements. It can also add and delete elements quickly, because, unlike an AVL tree, it does not need to be rebalanced. The efficiency of a hash-table implementation is largely dependent on the hash function implementation.

You cannot use a hash-table implementation variant when you require your elements to appear in main storage in sorted order (where elements earlier in the sorting order have lower addresses than elements later in the sorting order). On the other hand, you must use a hash table if you have a complex key (one composed, for example, of several attributes of an element), and either you cannot find a reasonable way to compare keys, or the comparison would be expensive.

For collections that do not provide access by key, but that support a hash-table implementation variant, the complete element is used as the input to the hash function.

Hashing, as implemented in the collection classes, allows elements to be stored in a potentially unlimited space, and therefore imposes no limit on the size of the collection. The figure below shows a hash-table implementation variant.

Hash-Table Implementation Variant



The hash function that calculates the index 3 from abcd is implemented as follows:

1. Each character is transformed into an integer according to its position in the alphabet.
2. The resulting integers are added together.
3. The result is divided by the hash table size. The remainder is the hash.

The principal behind a hash table is that the possibly infinite set of elements in your collection is partitioned into a finite number of hash values (1, 2, 3, ...). Your hash function is called with a key and a modulo value, and you use the key and the modulo value to arrive at an integer hash value. If for two different keys the hash function returns the same hash value (as for xyz and yyy in the previous

figure), a hash collision occurs. In such cases, a hash implementation constructs a collision list where all keys returning the same hash value are linked.

In the best case, for each different key, your hash function should return a different hash value. At the very least, it is desirable for the collision lists to remain small so that access time is fast. This means that hash values should be evenly distributed. Your hash function should randomly hash the key so that the hash value is not dependent on the key value in any trivial way. Your hash function should always return the same hash value for a given key and modulo provided to it.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Types of Collections
Possible Implementation Paths
"Class Template Naming Conventions" on page 13
"AVL Tree" on page 14
"B* Tree" on page 16
"Diluted Table" on page 17
"List"
"Table" on page 20

RELATED TASKS

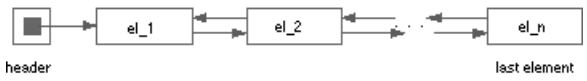
"Choose One of the Provided Implementation Variants" on page 23
"Chapter 2. Instantiate the Collection Classes" on page 25

List

A list uses pointers to link each element to its predecessor and successor. This implementation does not require contiguous memory for storing an array, which means that elements do not have to be shifted to make room for new elements or to close up gaps created by deleted elements.

Because storage is dynamically allocated and freed, this implementation variant is a good choice in applications that add or delete many elements, particularly where you cannot predict the amount of storage required. The figure below shows a list implementation variant.

List Implementation Variant



RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Types of Collections
Possible Implementation Paths
"Class Template Naming Conventions" on page 13
"AVL Tree" on page 14
"B* Tree" on page 16
"Diluted Table" on page 17
"Hash Table" on page 18
"Table" on page 20

RELATED TASKS

"Choose One of the Provided Implementation Variants" on page 23
"Chapter 2. Instantiate the Collection Classes" on page 25

Table

A table is an array implementation of a sequence. The elements are stored in contiguous cells of an array. In this representation, a list can easily be traversed, and new elements can easily be added to the tail of the list. If an element needs to be inserted into the middle of the list, however, all following elements need to be shifted to make room for the new element. Similarly, if an element needs to be removed from the list, and the element is not the last element in the list, all elements following the element to be deleted must be shifted in to close up the gap.

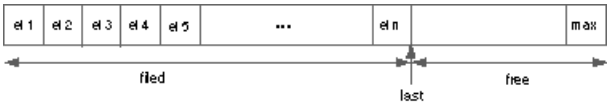
A table can access all elements quickly because all elements can be stored in a single storage block. If all of the following conditions hold true for your use of a collection, a table is a suitable implementation variant to use:

- The elements to be stored are small.
- You can predict with some accuracy how many elements your application will have to handle.
- Few or no elements will need to be added or deleted once the collection is first created.

Note that memory is statically allocated for tables, at the beginning of your program.

The figure below shows a table implementation variant.

Table Implementation Variant



RELATED CONCEPTS

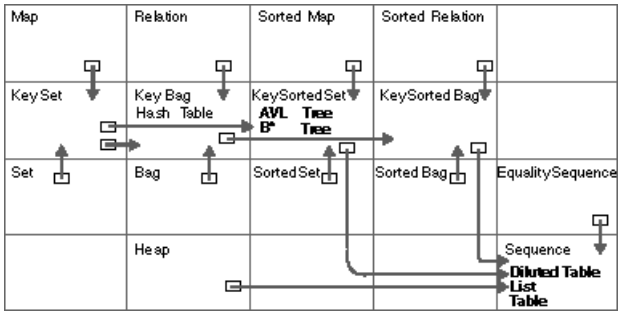
"Chapter 1. Collection Classes Overview" on page 1
Types of Collections
Possible Implementation Paths
"Class Template Naming Conventions" on page 13
"AVL Tree" on page 14
"B* Tree" on page 16
"Diluted Table" on page 17
"Hash Table" on page 18
"List" on page 19

RELATED TASKS

"Choose One of the Provided Implementation Variants" on page 23
"Chapter 2. Instantiate the Collection Classes" on page 25

Possible Implementation Paths

The figure below lists the basic and based-on implementations provided by the Collection Classes. The upper left corner of each cell contains the name of the (abstract) collection class; basic implementations are written in smaller letters in bold face, while based-on implementations are described by arrows starting from the class that they implement and ending in the (abstract) class on which they are based. An implementation choice for a given class must use either a basic implementation for this class or follow a based-on implementation path that ultimately leads to a basic implementation.



Take the example of the Set abstraction. The Set is not implemented directly. (You can tell this from the figure because no implementation variant name appears in bold in the box containing Set.) To determine the possible implementation variants for Set, follow the arrows out of the Set box:

- One arrow leads to the KeySet box. The KeySet box contains an implementation variant, **Hash Table**, so this is one possibility. An arrow also points from the KeySet Box to the KeySortedSet box, which allows the following possibilities:
  - **AVL Tree** (appears in KeySortedSet box)
  - **B* Tree** (appears in KeySortedSet box)
  - An arrow leads from KeySortedSet to Sequence, which contains the following implementation variants:
    - **List**
    - **Table**
    - **Diluted Table**

A Set can therefore be implemented using any of the six implementation variants cited in bold face above.

The following tables describes the Implementation variants provided for each flat collection. The letter “D” identifies the default implementations; “I” identifies implementation variants. If the space is blank, the feature is not supported.

Implementation variant	**Bag**	**Sorted Bag**	**Key Bag**	**Key Sorted Bag**
AVL Tree				
B* Tree				
Hash Table	I		D	
List	D	D	I	D
Table	I	I	I	I

Implementation variant	Bag	Sorted Bag	Key Bag	Key Sorted Bag
Diluted Table	I	I	I	I

Implementation variant	Set	Sorted Set	Key Set	Key Sorted Set
AVL Tree	D	D	D	D
B* Tree	I	I	I	I
Hash Table	I		I	
List	I	I	I	I
Table	I	I	I	I
Diluted Table	I	I	I	I

Implementation variant	Map	Sorted Map	Relation	Sorted Relation
AVL Tree	D	D		
B* Tree	I	I		
Hash Table	I		D	
List	I	I	I	D
Table	I	I	I	I
Diluted Table	I	I	I	I

Implementation variant	Sequence	Equality Sequence	Heap
AVL Tree			
B* Tree			
Hash Table			
List	D	D	D
Table	I	I	I
Diluted Table	I	I	I

RELATED CONCEPTS

Collection Classes Overview
Examples of Using Collection Classes

RELATED TASKS

Choose One of the Provided Implementation Variants

RELATED REFERENCES

IBag
ISortedBag

IKeyBag
IKeySortedBag
IHeap
IStack
IMap
ISortedMap
IQueue
IDeque
IPriorityQueue
IRelation
ISortedRelation
ISortedSet
IKeySet
IKeySortedSet
ISequence
IEqualitySequence

Choose One of the Provided Implementation Variants

When you are developing a program that uses a collection, you should begin by using the default implementation and go on to a final tuning phase where you choose implementations according to the actual requirements of your application. You can determine these requirements by profiling or by using other measurement tools. You need to choose between a variety of implementations provided by the Collection Classes as well as how to create your own implementation classes.

The collection implementations can be based on other collection classes. The based-on concept provides a systematic framework for choosing the most appropriate implementations. It is also useful for extending the Collection Classes with other basic implementations, such as specific kinds of search trees, and for using these implementations as the basis for other data abstractions such as sets, maps, and bags.

You can implement a given collection type (bag, key sorted set, etc.) in a number of different ways. The Collection Classes provide multiple implementation variants for collections because different variants have different performance and storage use characteristics. After you have coded and debugged an application that uses the Collection Classes, you can change an implementation to a variant that is well-suited to the ways in which you use the collection. For example, six implementation variants are listed for key set, including the default key set. These variants are implemented using the following concrete techniques:

- Sequences
  - List
  - Table
  - Diluted table
- Trees
  - AVL tree (the technique used for the default key set)
  - B* tree
- Hash table

As it turns out, the implementation variants for key set encompass all the concrete techniques used by the Collection Classes. Other collections may only use some of

the techniques in the list above. If you want to choose the best implementation variant for your program, you need to know the advantages of each concrete technique.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Possible Implementation Paths
"Class Template Naming Conventions" on page 13
"AVL Tree" on page 14
"B* Tree" on page 16
"Diluted Table" on page 17
"Hash Table" on page 18
"List" on page 19"Table" on page 20

RELATED TASKS

"Chapter 2. Instantiate the Collection Classes" on page 25

Replace the Default Implementation

By using typedefs to define the collection classes you use, you can easily replace the default implementation with another implementation. Suppose that you have a key set class called MyType that has been defined with the default implementation IKeySet. The definition of this class would look like this:

```
typedef IKeySet<Element, Key> MyType;
```

If you want to replace the default implementation, which uses an AVL tree, with a hash table implementation, you can replace the above implementation with the following definition:

```
typedef IHashKeySet<Element, Key> MyType;
```

If you replace a collection's default implementation with one of its implementation variants, you must determine what element functions and key-type functions need to be provided for the variant. You must then provide those functions. The list of required functions is not always the same for a collection's default implementation as for particular implementation variants.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes
Types of Collections
Possible Implementation Paths

RELATED TASKS

"Chapter 4. Copy and Reference Collections" on page 33

Chapter 2. Instantiate the Collection Classes

To use a collection class, you normally follow these three steps:

1. Instantiate a collection class template and provide arguments for the formal template arguments.
2. Define one or more objects of this instantiated class, possibly providing constructor arguments.
3. Apply functions to these objects.

The following example describes instantiation for the default implementation. Consider the following example header file for a class Person:

```
//person.h - Header file containing class Person
#include <iostream.h>
#include <istring.hpp>
class Person
{
    IString PersonName;
    IString TNumber;
public:
    //constructor
    Person () : PersonName(""), TNumber("") {}
    //copy constructor
    Person(IString Name, IStringNumber)
        : PersonName(Name), TNumber(Number)
    {
    }
    IString const& GetPersonName() const { return PersonName; }
    IString const& GetTNumber() const { return TNumber; }
    IBoolean operator==(Person const& A) const
    {
        return (PersonName == A.GetPersonName())
            && (TNumber==A.GetTNumber());
    }
    IBoolean operator<(Person const& A) const
    {
        return (PersonName < A.GetPersonName());
    }
};
```

For a given class, such as ISet, and a given element type, such as a class named Person, the instantiation for a new class that represents sets of persons could look like this:

```
//main.cpp - main file
#include <iset.h>
#include <iostream.h>
#include "person.h"   //person.h from the previous example
typedef ISet<Person> AddressList;
void main()
{
    AddressList Business;
    Person A("Sarah Vandewater", "90210");
    Business.add(A);
    cout << "\nThe set now contains "
         << Business.numberOfElements() <<" entries!\n";
}
```

Once the AddressList collection is defined, you can define AddressList objects Family, Business, and Sportclub as follows:

```
AddressList Family, Business, Sportclub;
```

You can also define the objects without introducing a new type name (AddressList):

```
ISet<Person> Family, Business, Sportclub;
```

However, you should begin by explicitly defining a named class, such as AddressList, that uses the default implementation. It is then easier to replace the default implementation with a better implementation later on.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1

Implement Bounded Collections

In the current implementation of the Collection Classes, all collections are unbounded. The concept of bounded collections is supported so that you can create your own bounded collection implementations. There are no bounded collections in the Collection Classes.

A bounded collection limits the number of elements it can contain. When a bounded collection contains the maximum number of elements (its bound), the collection is said to be full. This condition can be tested by the function isFull. If elements are added to a full collection, the exception IFullException is thrown. This behavior is useful for collections that are to have their storage allocated completely on the runtime stack.

You can determine the maximum number of elements in a bounded collection by calling the function maxNumberOfElements(). You can only call this function if the collection is bounded. You can determine whether a collection is bounded by calling the function isBounded().

The functions isBounded() and isFull() always return false. The function maxNumberOfElements() throws the exception INotBoundedException.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
"Chapter 10. Collection Class Library Exceptions" on page 71

RELATED TASKS

"Chapter 2. Instantiate the Collection Classes" on page 25

Chapter 3. Addition, Removal, and Replacement of Elements

Adding Elements
The add function places the element identified by its argument into the collection. It has two general properties:

- All elements that are contained in the collection before an element is added are still contained in the collection after the element is added.
- The element that is added will be contained in the collection after it is added.

Operations that contradict these properties are not valid. You cannot add an element to a map or sorted map that has the same key as an element that is already contained in the collection, but is not equal to this element (as a whole). In the case of a map and sorted map, an exception is thrown. Note that both map and sorted map are unique collections. The functions locateOrAddElementWithKey and addOrReplaceElementWithKey specify what happens if you try to add an element to a collection that already contains an element with the same key.

The figure below shows the result of adding a series of four elements to a map, a relation, a key set, and a key bag. The elements are pairs of a character and an integer. The character in the pair is the key. An element equality relation, if defined, holds between two elements if both the character and the integer in each pair are equal. The first row shows what each collection looks like after the element <a,1> has been added to each collection. Each following row shows what the collections look like after the element in the leftmost column is added to each.

Behavior of add for Unique and Multiple Collections

add	Map or sorted map	Relation or sorted relation	Key set or key sorted set	Key bag or key sorted bag
<a,1>	<a,1>	<a,1>	<a,1>	<a,1>
<b,1>	<a,1>, <b,1>	<a,1>, <b,1>	<a,1>, <b,1>	<a,1>, <b,1>
<a,1>	<a,1>, <b,1>	<a,1>, <b,1>	<a,1>, <b,1>	<a,1>, <b,1>, <a,1>
<a,2>	exception: Key Already Exists	<a,1>, <b,1>, <a,2>	<a,1>, <b,1>	<a,1>, <b,1>, <a,1>, <a,2>

The add function behaves differently depending on the properties of the collection:

- In unique collections, an element is not added if it is already contained in the collection.
- In sorted collections, an element is added according to the ordering relation of the collection.
- In sequential collections, an element is added to the end of the collection.

For sequential collections, elements can be added at a given position using add functions other than add, such as addAtPosition, addAsFirst, and addAsNext. Elements after and including the given position are shifted. Positions can be specified by a number, with 1 for the first element, by using the addAtPosition function. Positions can also be specified relative to another element by using the

addAsNext or addAsPrevious functions, or relative to the collection as a whole by using the addAsFirst or addAsLast functions.

Removing Elements
In the Collection Classes, you can remove an element that is pointed to by a given cursor by using the removeAt function. All other removal functions operate on the model of first generating a cursor that refers to the desired position and then removing the element to which the cursor refers. There is an important difference between element values and element occurrences. An element value may, for non-unique collections, occur more than once. The basic remove function always removes only one occurrence of an element.

For collections with key equality or element equality, removal functions remove one or all occurrences of a given key or element. These functions include remove, removeElementWithKey, removeAllOccurrences, and removeAllElementsWithKey. Ordered collections provide functions for removing an element at a given numbered position. Ordered collections also allow you to remove the first or last element of a collection using the removeFirst or removeLast functions.

After you have removed one element with the property, the entire collection would have to be searched for the next element with the property. Because all cursors of the collection become undefined when elements are removed, removing all elements with a given property from a collection cannot be done efficiently using cursors. If you want to remove all of the elements in a collection that have a given property, you should use the function removeAll and provide a predicate function as its argument. This predicate function has an element as argument and returns a value of type bool. The bool result tells whether the element is to be removed.

Sometimes you may want to pass more information to the predicate function. You can use an additional argument of type void*. The pointer then can be used to access a structure containing further information.

Replacing Elements
It is possible to modify collections by replacing the value of an element occurrence. Adding and removing elements usually changes the internal structure of the collection. Replacing an element leaves the internal structure unchanged. If an element of a collection is replaced, the cursors in the collection do not become undefined.

For collections that are organized according to element properties, such as an ordering relation or a hash function, the replace function must not change this element property. For key collections, the new key must be equal to the key that is replaced. For non-key collections with element equality, the new element must be equal to the old element as defined by the element equality relation. The key or element value that must be preserved is called the positioning property of the element in the given collection type.

Sequential collections and heaps do not have a positioning property. Element values in sequences and heaps can be changed freely. Replacing element values involves copying the whole value. If only a small part of the element is to be changed, it is more efficient to use the elementAt access function . The replaceAt function checks whether the replacing element has the same positioning property as the replaced element. (See “Chapter 10. Collection Class Library Exceptions” on page 71 for more details on preconditions.) When you use the elementAt function to replace part of the element value, this check is not performed. If you want to

ensure safe replacement (a replacement that does not change the positioning property), use replaceAt rather than elementAt.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Types of Collections
"Chapter 10. Collection Class Library Exceptions" on page 71

RELATED TASKS

"Chapter 2. Instantiate the Collection Classes" on page 25
"Add an Element to a Collection""Remove an Element from a Collection"

Add an Element to a Collection

To add an element to a collection, call the add function. Consider the following example:

```
// main.cpp - main file
#include <iset.h>
#include <iostream.h>
#include "person.h"   //person.h from the previous examples
typedef ISet<Person> AddressList;
void main()
{
    AddressList Business;
    Person A("Peter Black","714-50706");
    Person B("Carl Render","714-540321");
    Person C("Sandra Summers","214-660012");
    Business.add(A);
    Business.add(B);
    Business.add(C);
    Business.add(A);            //Person A is added for the second time
    cout << "\nThe set now contains " << Business.numberOfElements()
         <<" entries!\n";
}
```

If you run the program, the set will only contain 3 different entries. In a set, each element is unique. No two elements can be the same. To illustrate the difference between sets and bags, run the program using a bag rather than a set.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Addition, Removal, and Replacement of Elements

RELATED TASKS

"Remove an Element from a Collection"

Remove an Element from a Collection

Consider the following example:

```
//main.cpp - main file
#include <iset.h>
#include <iostream.h>
#include "person.h" //person.h from the previous examples
typedef ISet <Person> AddressList;
IBoolean noPhone(Person const& P,void*) //predicate function
{
```

```
    return P.GetTNumber()=="x";
}
void main()
{
    AddressList business;
    Person A("Peter Black","714-50706");
    Person B("Carl Render","714-540321");
    Person C("Sandra Summers","x");
    Person D("Mike Summers","x");
    business.add(A);
    business.add(B);
    business.add(C);
    business.add(D);
    business.add(A); //Person A is added for the second time
    cout << "\nThe set now contains " << business.numberOfElements()
         <<" entries!\n";
    business.removeAll(noPhone); //Person B is removed from the set
    cout << "\nThe set now contains " << business.numberOfElements()
         <<" entries!\n";
}
```

If you run the program, the set will only contain 2 elements as a result of the the remove function. Try modifying the program so that all persons with a telephone number are removed when the program is run.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
"Chapter 3. Addition, Removal, and Replacement of Elements" on page 27

RELATED TASKS

"Add an Element to a Collection" on page 29

Add and Overload Member Functions

Typically you will not derive from any of the Collection Classes. When you must derive classes from the Collection Classes be aware of the following:

1. The derived class only adds new member functions
2. The derived class overloads existing member functions. The derived collection class will not be used in polymorphic way.

Collection classes do not have virtual functions. You cannot override the member functions of a collection class.

For example, suppose you want to implement a set of integers that can give you information about the sum of integers contained in the collection. You create a class IntSet that is derived from ISet<int>. This class does the following:

1. Introduces the data member ivSum to hold the current sum.
2. Adds the member function sum, which returns the current sum.
3. Overloads the add member function so that it updates ivSum each time an integer is added to the collection.

In a real application, any add, replace or remove member function would have to be overloaded in order to update the sum of integers. For simplicity, this is not done in the example below:

```
#include <iset.h>
class IntSet: public ISet<int> {
    typedef ISet<int> Inherited;
public:
```

```
    IntSet(INumber n = 100)
        : ISet<int> (n), ivSum (0)
    {
    }
    bool add(int const& i)
    {
        ivSum += i;
        return Inherited::add(i);
    }
    int sum() const
    {
        return ivSum;
    }
private:
    int       ivSum;
};
//...
IntSet anIntSet;
anIntSet.add(1);
anIntSet.add(2);
cout << anIntSet.sum () << endl;
```

The output of this program is 3.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes

RELATED TASKS

"Example: Abstract Class Hierarchy"

Example: Abstract Class Hierarchy

The following example defines a universal printer class that accepts an arbitrary collection of jobs and prints their IDs. The elements are printed in the iteration order that is defined for the given collection. The key set running can be used as argument to the universal printer.

```
class JobPrinter {
public:
    print (IACollection <Job*> const& jobs)
    {
        cout << "ID     ..."
        ICursor *cursor = jobs.newCursor ();
        cout << "{ ";
        forICursor (*cursor) {
            cout << jobs.elementAt (*cursor)->id() &l2. ' ';
        }
        cout << "}\n";
        delete cursor;
    }
};
// ...
typedef IKeySet <Job*, JobId> JobSet;
JobSet running;
// ...
JobPrinter jobPrinter;
jobPrinter.print(running);
```

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes

RELATED TASKS

"Chapter 4. Copy and Reference Collections" on page 33"Add and Overload Member Functions" on page 30

Chapter 4. Copy and Reference Collections

The Collection Classes implement no structure sharing between different collection objects. The assignment operator and the copy constructor for collections are defined to copy all elements of the given collection into the assigned or constructed collection. You should remember this point if you are using collection types as arguments to functions. If the argument type is not a reference or pointer type, the collection is passed by the copy constructor, and changes made to the collection within the called function do not affect the collection in the calling function.

If you want a function to modify a collection, pass the collection as a reference:

```
void removeListMember (AddressList aList) { /* ... */ }   // wrong
void removeListMember (AddressList & aList) { /* ... */ } // right
```

For the sake of efficiency, avoid having a collection type as the return type of a function:

```
AddressList f()
{
    AddressList aList;
    // ...
    return aList;
}
Business=f();  //Very inefficient
```

In this program Business becomes a reference argument to the assignment operation, which would again copy the set. A better approach is:

```
void f(AddressList& aList) { /* ... */ }
// ...
f(Business);
```

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes

RELATED TASKS

"Replace the Default Implementation" on page 24

Chapter 5. Implement Element- and Key-Type Functionality

The member functions of the Collection Class Library call other functions to manipulate elements and keys. These functions are called element functions and key-type functions, respectively.

Member functions of the Collection Class Library may, for example, use the element's assignment or copy constructors for adding an element, or they may use the element's equality operator for locating an element in the collection. In addition, Collection Class functions use memory management functions for the allocation and deallocation of dynamically created internal objects (such as nodes in a tree or a linked list).

A given collection may require the following element functions:

- Default and copy constructor
- Destructor
- Assignment operator
- Equality test
- Ordering relation
- Key access
- Hash function

This list is the superset of all element functions and key-type functions that a Collection Class can ever require. For example, a collection without keys does not require any key-type functions, and a collection without element equality does not require an equality test.

A given collection may require the following key-type functions:

- Equality test
- Ordering relation
- Hash function

Where possible, these functions are already defined by the Collection Class Library. Default memory management functions are provided for usage with any element and key type. For the standard C++ data types int and char*, defaults are offered for all element and key-type functions. For all other element and key types, you must provide these functions.

For implementation variants where both equality test and ordering relation are required element functions (or where both are required key-type functions), the library does not define which of the two is used to determine element or key equality.

You can define these functions in three ways:

- Defining member functions of the element object type.
- Defining separate global functions.
- Using or defining an element operations class.

Some collections may require allocation and deallocation functions. The second and third methods can also be used to replace the default memory management functions for some of the collections.

Define Member Functions of the Element Object Type
The easiest way to provide the required element or key-type functions is to use member functions. For assignment, equality, and ordering relation, operator=, operator==, and operator< are used, respectively. Certain element functions and key-type functions must be defined as member functions. Others cannot be defined as member functions, but must be defined as separate functions.

You must define these functions using member functions:

- Constructors
- Destructors

Except for assignment, you must define member functions of a class as const. You will get a compile-time error if you do not include const in these definitions.

The following example shows how member functions must be defined as const:

```
class Element {
public:
    Element&       operator=  (Element const&);
    IBoolean       operator== (Element const&) const;
    IBoolean       operator<  (Element const&) const;
};
```

The Collection Class Library does not check or use the return type of operator=(). The return type of equality and ordering relation must be compatible with type IBoolean.

Define Separate Global Functions
You can use global functions to provide the required element and key functions. A global function is a function that is not a member of any class. Use global functions when, in instantiating the Collection Class, you have no control over the element class and the element class does not define the appropriate functions.

The following functions must be defined as global functions that are not members of any class:

- Functions for key access
- Functions for hashing
- Functions for memory management

The following shows what the declarations for these global functions must look like:

```
void          assign  (Element&, Element const&);
IBoolean      equal   (Element const&, Element const&);
long          compare (Element const&, Element const&);
Key const&    key     (Element const&);
unsigned long hash    (Element const&, unsigned long);
IBoolean      equal   (Key const&, Key const&);
long          compare (Key const&, Key const&);
unsigned long hash    (Key const&, unsigned long);
```

You can also use global functions for the standard memory management functions, as defined by the C++ language:

```
void*         operator new (size_t);
void          operator delete (void*);
```

The compare function must return a value that is less than, equal to, or greater than zero, depending on whether the first argument is less than, equal to, or greater than the second argument.

The hash function must return a value that is less than the second argument; this value may be achieved, for example, by computing the remainder (operator%) with the second argument. The hash function should evenly distribute over the range between zero and the second argument. For equal elements or keys, the hash element must yield equal results.

An efficient hash function is very important to the performance of your program.

For assign, equal, and compare, template functions are defined that will be instantiated unless otherwise specified. The default for assign uses the assignment operator, the default for equal uses the equality operator, and the default for compare uses two comparisons with operator<. It is therefore advisable to define your own compare function if the given element type has a more efficient implementation available. Such definitions are already provided for integer types using operator- and for char* using strcmp. By default, the standard memory management functions are used. (Using operator- works for integer types because the result of a-b can be used to determine whether a<b evaluates to true.)

The following examples demonstrate the use of a global function for the definition of the key access. The element class is Person, its data member PersonName is the key, and its member function GetPersonName is used to access the key.

Header File
The example below is the header file:

```
//person.h - header file containing class Person
#include <iostream.h>
#include <istring.hpp>
class Person {
    IString PersonName;    //This will be used as the key
    IString TNumber;
public:
    //constructor
    Person() : PersonName(""), TNumber("") {}
    //copy constructor
    Person(IString Name, IString Number)
        : PersonName(Name), TNumber(Number)
    {
    }
    IString const& GetPersonName() const { return PersonName; }
    IString const& GetTNumber() const { return TNumber; }
    IBoolean operator== (Person const& A) const
    {
        return (PersonName == A.GetPersonName()) &&
            (TNumber == A.GetTNumber());
    }
    IBoolean operator<(Person const& A) const
    {
        return (PersonName < A.GetPersonName());
    }
};
ostream& operator<<(ostream& os,Person A);
// Use separate function Key const& key (Element const&);
inline IString const& key (Person const& A)   //Key access
{
    return A.GetPersonName();
}
```

Main File
The example below is the main file.

```
//main.cpp - main file
#include "person.h"   //person.h from the previous example
#include <ikeyset.h>
typedef IKeySet <Person,IString> AddressList;
ostream& operator<<(ostream& os,Person A)
{
    return (os << endl << A.GetPersonName() << " " <<
        A.GetTNumber());
}
void main()
{
    AddressList Business;
    AddressList::Cursor myCursor(Business);
    Person A("Peter Black","714-50706");
    Person B("Carl Render","714-540321");
    Person C("Sandra Summers","x");
    Person D("Mike Summers","x");
    Business.add(A);
    Business.add(B);
    Business.add(C);
    Business.add(D);
    Business.removeElementWithKey("Carl Render");
    forICursor(myCursor) {
        cout<<Business.elementAt(myCursor);
    }
}
```

Use or Define an Element Operation Class

You can use element operation classes in cases where you want to place elements of one type into more than one collection, and where the element or key-type functions are different for each collection. For example, suppose you require an element type that is used to instantiate employee records that can be sorted either by name or by salary. You can declare an element class Person, and then place references to each Person instance into each of two collections. In one collection, the key is the name; in the other, the key is the salary. In your program, you need to define different element and key-type functions for hashing, comparison, and so on. Because these functions are not identical for both collections, you cannot define them within the class Person.

You can provide different sets of element and key-type functions for a given element type and multiple collections, by using the IG... class template for the collection you want to use. This class template lets you define element functions separately from the element class. In the case of the employee program, you can declare two classes as follows:

```
IGKeySortedSet <PersonPtr, int, SalaryOps> SalaryKSet;
IGKeySortedSet <PersonPtr, IString, NameOps> NameKSet;
```

You then need to define two other classes, SalaryOps and NameOps, which must contain appropriate element and key-type functions.

When you do not provide element or key operations by using an IG... collection, the standard class template (I... as opposed to IG...) defines default operations. These default operations are declared in istdops.h.

The following excerpt shows the definition of the class templates for ISequenceAsList and IGSequenceAsList:

```
template < class Element, class ElementOps >
class IGSequenceAsList { /* ... */ };
template < class Element >
class ISequenceAsList
```

```
    : public IGSequenceAsList<Element, IStdOps<Element>>
{
    /* ... */
};
```

The advantage of passing the arguments using an extra class instead of passing them as function pointers is that the class solution allows inlining.

The following is a skeleton for operation classes. The keyOps member must only be present for key collections. Note that all element and key operations must be defined as const.

```
template <class Element, class Key>
class ...Ops {
public:
    void*         allocate     (size_t) const;
    void          deallocate   (void*) const;
    void          assign       (Element&, Element const&) const;
    IBoolean      equal        (Element const&, Element const&) const;
    long          compare      (Element const&, Element const&) const;
    Key const&    key          (Element const&) const;
    unsigned long hash         (Element const&, unsigned long) const;
    class KeyOps
    {
         IBoolean      equal        (Key const&, Key const&) const;
         long          compare      (Key const&, Key const&) const;
         unsigned long hash         (Key const&, unsigned long) const;
    } keyOps;
};
```

You can inherit from the following class templates when you define your own operation classes. Templates with argument type T can be used for both the element and the key type.

```
class IStdMemOps {
public:
    void* allocate (size_t) const;
    void deallocate (void*) const;
};
template < class T >
class IStdAsOps
{
    void assign (T&, T const&) const;
};
template < class T >
class IStdEqOps
{
    IBoolean equal (T const&, T const&) const;
};
template < class T >
class IStdCmpOps
{
    long compare (T const&, T const&) const;
};
template < class Element, class Key >
class IStdKeyOps
{
    Key const& key (Element const&) const;
};
template < class T >
class IStdHshOps
{
    unsigned long hash (T const&, unsigned long) const;
};
```

The file istdops.h defines the above templates. It also defines other templates that combine the properties of one or more of the templates.

Things to Watch Out For
One of the C++ language rules states that function template instantiations are considered before conversions. Because the Collection Classes define default templates for element functions, functions such as equal or compare, defined for a class, will not be considered for that class's derived classes; the default template functions will be instantiated instead. In the following example, the compiler would attempt to instantiate the template compare function for class B, instead of inheriting the compare function of class A and converting B to A:

```
class A {
    // ...
};
long compare(A const&, A const&);
class B : public A {
    // ...
};
ISortedSet<B> BSet;
```

The instantiated default compare function for class B uses the operator< of B. If no operator< for B can be found, a compilation error occurs. You must define standard functions such as equal or compare for the actual element type B to prevent the template instantiation of those functions, in case you want to provide a class-specific equal or compare function for B.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes
"Collection Characteristics" on page 2

RELATED TASKS

"Define Equality Relation"
"Define Key or Element Equality" on page 41
"Define an Operations Class" on page 42
"Manage Memory with Element Operation Classes" on page 44

Define Equality Relation

A flat collection can have an equality relation defined for its elements. The default equality relation is based on the element as a whole, not just on one or more of its data members (for example, the key). For two elements to be equal, all data members of both elements must be equal. The equality relation is needed for functions such as those that locate or remove a given element. A flat collection that has an equality relation has element equality.

For collections containing non-built-in types, you can define your own equality relation to behave differently. For example, your equality relation could test only certain data members of two elements to determine element equality. In such cases, element equality may apply to two elements even when the elements are not exactly equal.

The equality relation for keys may be different than the equality relation for elements. Consider, for example, a job control block that has a priority and a job identifier that defines equality for jobs. You could choose to implement a job collection as unordered, with the job ID as key, or as sorted by priority, with the

priority as key. In the first case, you have fast access through the job ID but not through the priority; in the second case, you have fast access through the priority but not through the job ID. The ordering relation on the priority key in the second case does not yield a job equality, because two jobs can have equal priorities without being the same.

```
typedef unsigned long JobId;
typedef int Priority;
class Job {
     JobId ivId;           // These are private data members.
     Priority ivPriority;
public:
     JobId   id ()  const { return ivId; }
     Priority priority () { return ivPriority; }
};
// If ivId is the key:
JobId const& key (Job const& t)
{ return t.id (); }
// If ivPriority is the key:
Priority const& key (Job const& t)
{ return t.priority (); }
// ...
```

Functions like locateElementWithKey use the equality relation on keys to locate elements within a collection. A collection that defines key equality may also define element equality. Functions that are based on equality (such as locate) are only provided for collections that define element equality. Collections that define neither key equality nor element equality, such as heaps and sequences, provide no functions for locating elements by their values or testing for containment. Elements can be added and retrieved from such collections by iteration. For sequences, elements can also be added and retrieved by position.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes
"Collection Characteristics" on page 2

RELATED TASKS

"Chapter 5. Implement Element- and Key-Type Functionality" on page 35
"Define Key or Element Equality"
"Define an Operations Class" on page 42
"Manage Memory with Element Operation Classes" on page 44

Define Key or Element Equality

A sorted collection must define either key equality or element equality. A sorted collection that does not have a key defined must have an ordering relation defined for the element type. This relation implicitly defines element equality.

Keys can be used to access a particular element in a collection. The alternative to defining element equality as equality of all data members is to define it as equality of keys only. (In the example below, this means defining job equality as equality of the job ID.) Use this alternative only when you are sure that keys are unique. When you use this alternative, you can locate an element only with the key (using locateElementWithKey(key) instead of locate(element). Locating elements by key improves performance, particularly if the complete element is large or difficult to construct in comparison to the key alone. Consider the two alternatives in the following example:

```
// First solution
JobId const& key (Job const& t) {  return t.id;  }
KeySet < Job, int > jobs;
// ...
jobs.locateElementWithKey (1);
// Second solution
IBoolean operator== (Job const& t1, Job const& t2)
{  return t1.id == t2.id;  }
Set < Job > jobs;
// ...
Job t1;
t1.id = 1;
jobs.locate (t1);
```

The first solution is superior, if job construction (Job (t1) requires a significant proportion of the total system resources used by the program.

The Collection Class Library provides sorted and unsorted versions of maps and relations, for which both key and element equality must be defined. These collections are similar to key set and key bag, except that they define functions based on element equality, namely union and intersection. The add function behaves differently toward maps and relations than it does toward key set and key bag.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes
"Collection Characteristics" on page 2

RELATED TASKS

"Chapter 5. Implement Element- and Key-Type Functionality" on page 35
"Define Equality Relation" on page 40
"Define an Operations Class"
"Manage Memory with Element Operation Classes" on page 44

Define an Operations Class

To define an operations class, use the predefined templates for standard functions, and define the specific functions individually.

Consider, for example, a person's name (PersonName) and phone number (TNumber). The name serves as the key for an address list, while the phone number serves as the key for a phone list. Because the key function is already defined to yield the person's name, the phone list has to be instantiated in the following way:

Header File
This is the header file:

```
//person.h - header file containing class Person
#include <iostream.h>
#include <istring.hpp>
#include <istdops.h>
class Person {
    IString PersonName;
    IString TNumber;
public:
    //constructor
    Person()
```

```
        : PersonName(""), TNumber("")
    {
    }
    //copy constructor
    Person(IString name, IString number)
        : PersonName(name), TNumber(number)
    {
    }
    IString const& GetPersonName() const { return PersonName; }
    IString const& GetTNumber() const { return TNumber; }
    IBoolean operator==(Person const& A) const
    {
         return (PersonName == A.GetPersonName()) &&
             (TNumber==A.GetTNumber());
    }
    IBoolean operator<(Person const& A) const
    {
        return (PersonName < A.GetPersonName());
    }
};
ostream& operator<<(ostream& os, Person A);
class PhoneOps : public IStdMemOps, public IStdAsOps<Person> {
public:
    IString const& key(Person const& A) const
    {
        return A.GetTNumber();
    }
    IStdCmpOps <IString> keyOps;
};
inline IString const& key(Person const& A)   //Key access
{
    return A.GetPersonName();
}
```

Main File

This is the main file:

```
//main.cpp - main file
#include "person.h"   //person.h from the previous example
#include <istdops.h>
#include <ikeyset.h>
typedef IKeySet <Person,IString> AddressList;
typedef IGKeySet <Person,IString,PhoneOps> PhoneList;
ostream& operator<<(ostream& os,Person A)
{
    return (os << endl << A.GetPersonName() <<
        " " << A.GetTNumber());
}
void main()
{
    AddressList Business;
    PhoneList PhoneBook;
    AddressList::Cursor myCursor1(Business);
    PhoneList::Cursor myCursor2(PhoneBook);
    Person A("Peter Black","714-50706");
    Person B("Carl Render","714-540321");
    Person C("Sandra Summers","x");
    Person D("Mike Summers","x");
    Business.add(A);
    Business.add(B);
    Business.add(C);
    Business.add(D);
    PhoneBook.add(A);
    PhoneBook.add(B);
    PhoneBook.add(C);
    PhoneBook.add(D);
    cout << "\n\nPhoneBook before removing an element: ";
```

```
    forICursor(myCursor2) {
        cout<<PhoneBook.elementAt(myCursor2);
    }
    cout << "\n\nPhoneBook after removing an element: ";
    PhoneBook.removeElementWithKey("714-50706");
    forICursor(myCursor2) {
        cout<<PhoneBook.elementAt(myCursor2);
    }
    cout << "\n\nBusiness before removing an element: ";
    forICursor(myCursor1) {
        cout<<Business.elementAt(myCursor1);
    }
    cout << "\n\nBusiness after removing an element: ";
    Business.removeElementWithKey("Peter Black");
    forICursor(myCursor1) {
        cout<<Business.elementAt(myCursor1);
    }
}
```

The functions that are required for a particular Collection Class depend not only on the abstract class but also on the concrete implementation choice. If you choose a set to be implemented through a hash table, the elements require a hash function. If you choose a (sorted) AVL tree implementation, elements need a comparison function. Even the default implementations may require more functions to be provided than would be necessary for the collection interface.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes
"Collection Characteristics" on page 2

RELATED TASKS

"Chapter 5. Implement Element- and Key-Type Functionality" on page 35
"Define Equality Relation" on page 40
"Define Key or Element Equality" on page 41
"Manage Memory with Element Operation Classes"

Manage Memory with Element Operation Classes

The following illustrates the use of memory management with element operation classes.

Suppose you want to use your own element operation class to provide a special form of memory management. For example, you want an entire collection (the collection body plus the elements) to reside in a database, or in shared memory. To do this you can code:

```
IGSequenceAsList<Element, MyOperationsClass>
```

where MyOperationsClass is an element operations class you have coded, which provides your own element allocate and deallocate operations. This class may or may not inherit from previously described template classes, except that it must inherit from IStdMemOps.

A certain instance of your collection is instantiated together with an instance of your MyOperationsClass. You can retrieve the this pointer of this instance of MyOperationsClass to find out where the collection is instantiated, and you can

use this address in your implementation of the allocate element function to allocate your elements in the same memory pool where your collection resides.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes
"Collection Characteristics" on page 2

RELATED TASKS

"Chapter 5. Implement Element- and Key-Type Functionality" on page 35
"Define Equality Relation" on page 40
"Define Key or Element Equality" on page 41
"Define an Operations Class" on page 42

Chapter 6. Iteration

Iterating over all or some of elements of a collection is a common operation. The Collection Classes give you two methods of iteration:

- Using cursors
- Using the allElementsDo function together with applicators or applicator functions

Ordered (including sorted) collections have a well-defined ordering of their elements, while unordered collections have no defined order in which the elements are visited in an iteration. However, in both cases, each element is visited exactly once.

You cannot add or remove elements from a collection while you are iterating over a collection, or all elements may not be visited once. You cannot use any of the iterations described in this section if you want to remove all of the elements of a collection that have a certain property. Instead, use the function removeAll that takes a predicate function as an argument.

Iteration with Cursors

A *cursor* is a reference to the position of an element in a collection. If the position of that element changes, the cursor is *invalidated*. A cursor is always associated with a collection. The collection is specified when the cursor is created. Each collection function that takes a cursor argument has a precondition that the cursor actually belongs to the collection. Simple functions, such as advancing the cursor, are functions of the cursor itself.

With cursors the Collection Classes provide:

- An iteration scheme that is simpler than using applicators.
- The ability to define functions that return cursors. Such functions can give you fast access to an element if it exists, or indicate the non-existence of an element by returning an invalid cursor.

Cursors are only temporarily defined. As soon as elements are added to or removed from the collection, existing cursors become *undefined*. When a cursor becomes undefined, one of the three following situations occur:

1. The cursor is invalidated (isValid will return false).
2. The cursor remains valid and points to an element of the collection; however, it may point to a different element than before.
3. The cursor remains valid but no longer points to an element of the collection.

Do not use an undefined cursor as an argument to a function that requires the cursor to point to an element of the collection. The Collection Classes do not provide any methods to determine whether or not a cursor is undefined.

Each concrete collection class, such as ISet<int>, has an inner definition of a class Cursor that can be accessed as ISet<int>::Cursor.

Because abstract classes declare functions on cursors just as concrete classes do, there is a base class ICursor for these specific cursor classes. To allow for the creation of specific cursors for all kinds of collections, every abstract class has a virtual member function newCursor, which creates an appropriate cursor for the given collection object.

Iteration with allElementsDo
Cursor iteration has two possible drawbacks:

- For unordered collections, the explicit notion of an (arbitrary) ordering may be undesirable for stylistic reasons. For example, it could mislead you (or another programmer) into perceiving or exploiting an order where in fact the order does not exist or is not guaranteed.
- Iteration in an arbitrary order might be done more efficiently using something other than cursors. For example, with tree representations, a recursive descent iteration may be faster than the cursor navigation, even though the time for extra function calls must be considered.

The Collection Classes provide the allElementsDo function that addresses both drawbacks by calling a function that is applied to all elements. The function returns a value of type bool that tells whether the iteration should be continued or not. For ordered collections, the function is applied in this order. Otherwise the order is unspecified.

The function that is applied in each iteration step can be given in two ways: directly as a C++ function, or by defining the function as a method of a user-defined applicator class:

- As a C++ function: Code the function that you want to be applied to all elements as a C++ function, then use allElementsDo to apply the function to the elements.
- As an object of an applicator class: Derive your own class from one of the *applicator classes*, IApplicator or IConstantApplicator. Redefine the virtual function applyTo(). When you call allElementsDo(), the program will apply the code in applyTo() to all the elements of your collection.

The second possibility is more flexible. You can add additional arguments to applyTo() if needed. You can also reuse the applicator class.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1

RELATED TASKS

"Use Cursors to Locate and Access Elements"
"Use Cursors to Iterate Over a Collection" on page 50
"Use allElementsDo and Applicators to Iterate Over a Collection" on page 51

Use Cursors to Locate and Access Elements

Cursors provide a basic mechanism for accessing elements of collection classes. For each collection, you can define one or more cursors, and you can use these cursors to access elements. Collection Class functions such as elementAt, locate and removeAt use cursors.

The elementAt function lets you access an element using a cursor. The elementAt function returns a reference to an element, thereby avoiding copying the elements. Suppose that an element had a size of 20KB and you want to access a 2-byte data member of that element. If you use elementAt to return a reference to this element, you avoid having to copy the entire element to a local variable.

Several other functions, such as firstElement or elementWithKey, return a reference to an element. They can be thought of as first executing a corresponding cursor function, such as setToFirst or locateElementWithKey, and then accessing the element using the cursor.

You must determine if the element exists before trying to access it. If its existence is not known from the context, it must first be checked.

To save the extra effort of locating the desired element twice (once for checking whether it exists and then for actually retrieving its reference), use the cursor that is returned by the locate function for fast element access:

```
//main.cpp - main file
#include <iset.h>
#include <iostream.h>
#include "person.h" //person.h from the previous examples
typedef ISet <Person> AddressList;
void main()
{
    AddressList business;
    AddressList::Cursor myCursor(Business); //Cursor definition
    Person A("Peter Black","714-50706");
    Person B("Carl Render","714-540321");
    Person C("Sandra Summers","x");
    Person D("Mike Summers","x");
    Person E;
    business.add(A);
    business.add(B);
    business.add(C);
    business.add(D);
    if (business.locate(B, myCursor)){
        E=business.elementAt(myCursor) ;
    }
    else {
        cout << "\nElement not in set !";
    }
    business.remove(B); //myCursor is no longer valid
    if (business.locate(B, myCursor)) {
        E=business.elementAt(myCursor);
    }
    else {
        cout << "\nElement not in set !";
    }
}
```

The elementAt function can also be used to replace the value of the referenced element. You must ensure that the positioning property of the element is not changed with respect to the given collection.

There are two versions of elementAt:

Element const& elementAt (ICursor const&) const
Element& elementAt (ICursor const&)

Use the first version of elementAt if you want to ensure that the located element cannot be changed by any subsequent function.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
"Chapter 6. Iteration" on page 47

RELATED TASKS

"Use Cursors to Iterate Over a Collection"
"Use allElementsDo and Applicators to Iterate Over a Collection" on page 51
"Use allElementsDo and Applicators to Iterate Over a Collection" on page 51"Handle Exceptions with Cursors" on page 74

Use Cursors to Iterate Over a Collection

Cursor iteration can be done with a for loop. Consider the following example:

```
//main.cpp - main file
#include <iset.h>
#include <iostream.h>
#include "person.h"   //person.h from the previous examples
typedef ISet <Person> AddressList;
ostream& operator<<(ostream& os,Person A)
{
    return (os << endl << A.GetPersonName() << " " <<
        A.GetTNumber());
}
void main()
{
    AddressList Business;
    AddressList::Cursor myCursor(Business);
    Person A("Peter Black","714-50706");
    Person B("Carl Render","714-540321");
    Person C("Sandra Summers","x");
    Person D("Mike Summers","x");
    Person E;
    Business.add(A);
    Business.add(B);
    Business.add(C);
    Business.add(D);
    //List of all elements in the set
    for (myCursor.setToFirst();
        myCursor.isValid();
        myCursor.setToNext())
    {
        cout << Business.elementAt(myCursor);
    }
}
```

AddressList::Cursor is the Cursor class that is nested within the class AddressList. Its constructor takes a Business object as an argument. The name of the cursor object in the example above is myCursor.

The Collection Classes define a macro forICursor that lets you write a cursor iteration even more elegantly:

```
#define forICursor(c)      \
    for ((c).setToFirst(); \
        (c).isValid();     \
        (c).setToNext())
```

Use it like this:

```
forICursor(myCursor) {
    cout << Business.elementAt(myCursor);
}
```

If the element is used read-only, a function of the cursor can be used instead of elementAt(myCursor):

```
forICursor(myCursor) {
    cout << myCursor.element();   //myCursor is associated to Business
}
```

The function element above is a function of the Cursor class. It returns a const reference to the element currently pointed at by the cursor. The element returned might therefore not be modified. Otherwise it would be possible to manipulate a constant collection by using cursors.

To remove multiple elements from a collection, use the removeAll function with a predicate function as an argument. Using cursor iteration to identify the elements to remove causes the first element removed to invalidate the cursor.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
"Chapter 6. Iteration" on page 47

RELATED TASKS

"Use Cursors to Locate and Access Elements" on page 48
"Use allElementsDo and Applicators to Iterate Over a Collection"
"Use allElementsDo and Applicators to Iterate Over a Collection""Handle Exceptions with Cursors" on page 74

Use allElementsDo and Applicators to Iterate Over a Collection

The following example shows the use of the allElementsDo function and applicators.

```
//main.cpp - main file
#include <iset.h>
#include <iostream.h>
#include "person.h"   //person.h from the previous examples
typedef ISet<Person> AddressList;
ostream& operator<<(ostream& os, Person A)
{
    return (os << endl << A.GetPersonName() << " " <<
        A.GetTNumber());
}
class ListApplicator: public IConstantApplicator<Person> {
public:
    IBoolean applyTo(Person const& A)
    {
        cout << A;
        return true;
    }
};
void ListFunction(AddressList const& List)
{
    ListApplicator LA;
    List.allElementsDo (LA);
}
void main()
{
    AddressList Business;
    AddressList::Cursor myCursor(Business);
    Person A("Peter Black","714-50706");
    Person B("Carl Render","714-540321");
    Person C("Sandra Summers","x");
    Person D("Mike Summers","x");
    Person E;
    Business.add(A);
```

```
    Business.add(B);
    Business.add(C);
    Business.add(D);
    //List of all elements in the set
    ListFunction(Business);
}
```

This time you get the address listing using an applicator.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
"Chapter 6. Iteration" on page 47

RELATED TASKS

"Use Cursors to Locate and Access Elements" on page 48
"Use Cursors to Iterate Over a Collection" on page 50
"Use allElementsDo and Applicators to Iterate Over a Collection" on page 51"Handle Exceptions with Cursors" on page 74

Chapter 7. Smart Pointers

In C++, variables and function arguments have their values copied when they are assigned. This copying can decrease a program's efficiency, especially when the objects are large. To improve efficiency, pointers or references are often used for common objects. For example, a pointer or reference to the object can be copied, instead of the object itself.

For efficiency, pointers to elements can be used as collection element types, rather than the elements themselves. References are not allowed as collection element types.

The Collection Classes define five pointer classes, called *smart pointers*. You can store these smart pointers, as well as standard C++ pointers, as elements in any collection:

- IElemPointer
- IAutoPointer
- IAutoElemPointer
- IMngPointer
- IMngElemPointer

Smart Element Pointers
IElemPointer, IAutoElemPointer and IMngElemPointer are special smart pointers that are designed to be kept as elements in a collection. If you store standard C++ pointers in a collection, the collection performs all element and key-type functions, except assignment, on the pointers themselves rather than on the object to which the pointer refers. This is not always what you intend. If you want the collections to perform element functions, such as equality test, on the referenced elements instead, use one of the smart element pointers (one of the classes named I*xxx*ElemPointer). They are objects which behave as pointers to the actual element. With the element pointers, the elements themselves are not stored in the collection, although information from the elements is used by Collection Classes functions.

Automatic Storage Management
IAutoPointer, IAutoElemPointer, IMngPointer and IMngElemPointer perform storage management, which means that under certain conditions they automatically delete the object to which they refer. Automatic pointers (IAutoPointer and IAutoElemPointer) automatically delete the referenced object when the automatic pointer instance is destructed or the automatic pointer is used to point to another object. Managed pointers (IMngPointer and IMngElemPointer) keep a reference count for each referenced object. The referenced object of a managed pointer is deleted only when the last managed pointer to the object is destructed.

IElemPointer does not provide automatic storage deallocation.

Automatic storage management is particularly useful when functions return pointers or references to objects that they have created (dynamically allocated), and the last user of the object is responsible for cleaning up.

To exploit the advantage of memory management, you can use non-element smart pointers (IAutoPointer and IMngPointer), instead of standard C++ pointers,

without storing them in a collection. They behave similarly to standard C++ pointers. For example, if you check the equality of two such pointers from your collection of pointers, true is only returned if the pointers point to the same address (this is the same behavior as you would expect for native C++ pointers).

The auto_ptr Template Class
The Standard Template Library (STL) provides the auto_ptr template class. This template class has the equivalent functionality of the IAutoPointer class, but does not have any dependencies on the IBM Open Class. The auto_ptr template class is defined in <memory>.

For more information about the auto_ptr template class refer to Bjarne Stroustrup's *The C++ Programming Language*, Chapter 14, "Exception Handling," section 14.4.2. This reference is included in the VisualAge C++ PDF files.

RELATED CONCEPTS

Collection Classes Overview

RELATED TASKS

Construct Smart Pointers
Choose the Appropriate Smart Pointer Class
Use Automatic Pointers
Use Element Pointers
Use Managed Pointers

RELATED REFERENCES

The C++ Programming Language: Exception Handling

Construct Smart Pointers

All smart pointers have two constructors: a default constructor that initializes the pointer to NULL, and a constructor taking a C++ pointer to an element that you must have created before (using new).

Implicit conversions from a C++ pointer to a managed or automatic pointer are dangerous: elements might be implicitly deleted without your being aware that this has happened. Therefore, the conversion functions for these classes take an extra argument IINIT to make the construction explicit. Hence, the notation for creating a managed or automatic pointer is:

```
IAutoPointer < E > ePtr (new E, IINIT);
```

After you have constructed a managed or automatic pointer from a C++ pointer, you should no longer use the C++ pointer. You should only access the element through the pointer of the given class. Otherwise, the element could be implicitly destructed while a C++ pointer still refers to it. In particular, you must not construct two managed pointers or two automatic pointers from the same C++ pointer, because this would cause the managed pointers to keep two separate reference counts, and to implicitly delete the referenced element twice. For example:

```
IString* s = new IString("...");
IMngPointer < IString > p1 (s, IINIT); // OK
IMngPointer < IString > p2 (s, IINIT); // NO!
// Do not use s a second time, because the compiler may try to
// delete the IString object referred to by s, p1, and p2 twice.
```

You should keep the following rule in mind when using managed or automatic pointers created from standard pointers: Never use the C++ pointer once the managed or automatic pointer has been created from it, because this may interfere with the automatic storage management. For example, the object that is referenced by a C++ pointer and by an automatic pointer created from this C++ pointer, is deleted as soon as the automatic pointer gets out of scope. The C++ pointer then points to undefined storage.

The extra IINIT argument is introduced to make such situations explicit and especially to avoid the usage of the constructor as an implicit conversion operator. The IINIT argument is defined as follows:

```
enum IExplicitInit {IINIT};
```

Basic Types
The smart pointers do not work with basic types such as int, long, and char.

Key Collections
If you implement a key collection containing element pointers, you must define your key function with the element as input, not the pointer to the element, for example:

```
typedef IKeySortedSet<IMngElemPointer<Element>,int> keySortedSetOfPointers;
int const& key(Element const& element)
{
    return element.elementKey();
}
```

where elementKey returns the element's key.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Smart Pointers

RELATED TASKS

"Choose the Appropriate Smart Pointer Class"
Use Automatic Pointers
"Use Element Pointers" on page 58
"Use Managed Pointers" on page 61

Choose the Appropriate Smart Pointer Class

The following features of Collection Classes pointer types give you the choices shown in the table below. (Standard C++ pointers are included for comparison.)

	Destruction of Pointed Objects		
	Not managed	When out-of-scope	Reference counted
Collections call element operations on pointer	Standard C++ pointer	IAutoPointer	IMngPointer
Collections call element operations on referenced object	IElemPointer	IAutoElemPointer	IMngElemPointer

Smart pointers can only take arguments of type class or struct. This is because the overloaded operator-> needs to return an object of such a type. You can apply pointer objects from these five classes in the same way you use ordinary C++ pointers, with the * and -> operators. Elements are implicitly deleted except in the case of IElemPointer. To delete an element referred to by an IElemPointer you must use an explicit conversion to the referenced element type:

```
IElemPointer<E> ptr;
// ...
delete (E*) ptr;
```

Element Functions and Elements Referenced by Pointers
If you want element functions to work on the elements referenced by the pointers, the Collection Classes offer the IElemPointer, IAutoElemPointer and IMngElemPointer pointer classes, which are instantiated with the element type. Pointers of these classes automatically apply all element functions, except for assignment, to the referenced object. Element pointers are constructed from C++ pointers. The C++ dereferencing operators * and -> are defined, for element pointers, to refer to the referenced objects.

The dynamically created elements are not automatically deleted when they are removed from the collection.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Smart Pointers

RELATED TASKS

"Construct Smart Pointers" on page 54
Use Automatic Pointers
"Use Element Pointers" on page 58"Use Managed Pointers" on page 61

Use Automatic Pointers

Automatic pointers should be used when the lifetime of the element is the same as the lifetime of the pointer, but when an explicit deletion of the element is awkward or even impossible. This applies in particular to pointers to objects that are dynamically created within a function, and whose lifetime is the scope of the function. The function may be left through several return statements or through an exception being thrown from some other function being called.

Assign One Automatic Pointer to Another
Using the assignment operator, the automatic pointer is used to point to another element (which is implicitly a new element). The assigned pointer is set to NULL.

Hold Automatic Pointers in a Collection
If you define a collection taking automatic pointers as elements, the elements are automatically deleted when the collection is destructed, when an element is removed, or, if the element was not added to the collection, when the variable or temporary holding the pointer is destructed. The following example deletes all pointers that were added previously to the set with the destruction of the set:

```
#include "person.h"   // Declaration of Person class
#include <istdops.h>
#include <iset.h>
typedef IAutoElemPointer<Person> AEPointer;
typedef ISet<AEPointer> AddressList;
```

```
ostream& operator<<(ostream& os,Person A)
{
   return (os << endl << A.GetPersonName() <<" "<<A.GetTNumber());
}
void main()
{
    AddressList Business;
    AddressList::Cursor myCursor1(Business);
    Business.add(AEPointer (new Person("Peter Black","714-50706"),IINIT));
    Business.add(AEPointer (new Person("Carl Render","714-540321"),IINIT));
    Business.add(AEPointer (new Person("Sandra Summers","x"),IINIT));
    Business.add(AEPointer (new Person("Mike Summers","x"),IINIT));
    //The temporary automatic pointer variables were set to NULL
    //when the pointer was copied to the collection.
    //The following deletes the second Person ("Sandra ..."), because it
    //was not added (note that in a set, each element occurs only once).
    {
        Business.add(AEPointer (new Person("Sandra Summers","x"),IINIT));
    }
    forICursor(myCursor1) {
        cout << *Business.elementAt(myCursor1);
    }
}
```

Transfer Automatic Pointers Between Functions
You should be aware of the implementation details described below when transferring automatic pointers between functions. Consider the following cases:

Case 1: A calling function passes an automatic pointer to a called function and the pointer is returned.

The following code passes an automatic pointer to a called function and the pointer is returned.

```
IAutoPointer<Int> someFunc(IAutoPointer<Int> autoIntPtr)
{
    return autoIntPtr;
}
...
main ()
{
    IAutoPointer<Int> myIntPtr(new Int(5), IINIT);
    cout << *someFunc(myIntPtr) << endl;
}
```

This program results in the following taking place at runtime:

- The main function constructs an IAutoPointer object autoIntPtr and initializes it with the address of Int object 5 (where Int is a class that wraps an int).
- On invocation of someFunc, the copy constructor of IAutoPointer is called and the new constructed auto pointer is initialized with the address of the given input pointer. The given pointer is set to NULL. On return from someFunc, the copy constructor of IAutoPointer constructs a new auto pointer in main and initializes it with the address of the auto pointer object returned from someFunc, which then is destructed.
- When main exits, it calls the destructors for all auto pointer objects and the destructor for Int object 5.

Case 2: A called function has no input, but returns an object that has been dynamically created using an automatic pointer.

The following code returns an object that has been dynamically created using an automatic pointer.

```
Int someFunc()
{
    IAutoPointer<Int> autoIntPtr(new Int(6), IINIT);
    return *autoIntPtr;
}
...
main()
{
    cout << someFunc() << endl;
}
```

This program results in the following taking place at runtime:

- On invocation of someFunc, this function constructs an IAutoPointer object, constructs an Int(6) object, and initializes the auto pointer with the address of Int(6).
- On return from someFunc, the copy constructor of Int constructs a new Int(6) object in main. The auto pointer object and the Int(6) object in someFunc are destructed.
- On exit from main, the Int(6) object is destructed.

Automatic Pointer Copy Constructor and Assignment Operator
An automatic pointer's copy constructor and assignment operator are defined in a way that resets the source pointer to NULL. This prevents multiple automatic pointers from pointing to the same element. In the following example, p2 is implicitly set to NULL:

```
IAutoPointer<SomeType> p1, p2;
...
p1 = p2;
```

However, the copy constructor and assignment operator still take a const argument (using a const cast-away) to maintain compliance with the standard interface for these operations. This standard interface is required, for example, when you use these types as element types in collections, because the copy constructor and assignment operator are required to have such an interface. (Otherwise, the collection's add function could not take a const argument.)

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Smart Pointers

RELATED TASKS

"Choose the Appropriate Smart Pointer Class" on page 55
"Construct Smart Pointers" on page 54
"Use Element Pointers""Use Managed Pointers" on page 61

Use Element Pointers

If you create a collection of C++ pointers or pointers of type IMngPointer or IAutoPointer, Collection Classes methods that use element comparison functions will do the comparison on the elements' pointers instead of on the elements themselves.

Directing Element Functions to the Referenced Element
The classes IElemPointer, IMngElemPointer, and IAutoElemPointer internally use a function called elementForOps to direct functions such as equal and compare to the referenced element, so that they are not applied to the pointer itself and so that

instantiations such as ISet <IElemPointer <Person>> perform the functions on the elements. This indirection is usually transparent but you must consider it when you derive classes from the IElemPointer class. The standard operation classes first apply a function elementForOps to the element before they apply the corresponding non-member (equal, ...) function. By default, a corresponding template function is instantiated for elementForOps which takes an element as input and returns that element. For pointer classes that perform operations on the pointers themselves (IAutoPointer, IMngPointer), this function takes the pointer as input and returns the same pointer. For pointer classes that perform the operations on the referenced elements (IElemPointer, IAutoElemPointer, IMngElemPointer), this function takes the pointer as input and returns the referenced element. If a class derived from IElemPointer<E> is used as a collection element type, the default template functions must be instantiated before a conversion will be considered. A derived class must therefore explicitly redefine the elementForOps function, as shown in the following example, where class PersonPtr redefines both versions of elementForOps by calling the default elementForOps with a PersonPtr as argument. Both versions are then made to return a cast to Person reference.

```
//person.h - header file containing class Person
#include <iostream.h>
#include <istring.hpp>
#include <iptr.h>
class Person
{
    IString PersonName; //This will be used as the key
    IString TNumber;
public:
    //constructor
    Person () : PersonName(""), TNumber("") {}
    //copy constructor
    Person(IString Name, IString Number)
        : PersonName(Name), TNumber(Number)
    {
    }
    IString const& GetPersonName() const { return PersonName; }
    IString const& GetTNumber() const { return TNumber; }
    IBoolean operator==(Person const& A) const
    {
        return (PersonName == A.GetPersonName()) &&
            (TNumber==A.GetTNumber());
    }
    IBoolean operator<(Person const& A) const
    {
        return (PersonName < A.GetPersonName());
    }
};
class PersonPtr : public IElemPointer<Person> {
    friend inline Person& elementForOps (PersonPtr& A)
    {
        return (Person&)elementForOps((IElemPointer<Person>&)A);
    }
    friend inline Person const& elementForOps(PersonPtr const& A)
    {
        return (Person const&)elementForOps((IElemPointer<Person>&)A);
    }
public:
    PersonPtr() : IElemPointer<Person>() {}
    PersonPtr(Person* ptr,IExplicitInit IINIT)
        : IElemPointer<Person>(ptr,IINIT) {}
};
ostream& operator<<(ostream& os,Person A);
```

```
inline IString const& key(Person const& A) //Key access
{
    return A.GetPersonName();
};
```

This is the main file.

```
//main.cpp - main file
#include "person.h"   //person.h from the previous example
#include <istdops.h>
#include <iset.h>
typedef ISet <PersonPtr> AddressList;
ostream& operator<<(ostream& os,Person A)
{
    return (os << endl << A.GetPersonName() << " " <<
        A.GetTNumber());
}
void main()
{
    AddressList Business;
    AddressList::Cursor myCursor1(Business);
    PersonPtr Aptr (new Person("Peter Black","714-50706"),IINIT);
    PersonPtr Bptr (new Person("Carl Render","714-540321"),IINIT);
    PersonPtr Cptr (new Person("Sandra Summers","x"),IINIT);
    PersonPtr Dptr (new Person("Mike Summers","x"),IINIT);
    PersonPtr CopyCptr (new Person("Sandra Summers","x"),IINIT);
    Business.add(Aptr);
    Business.add(Bptr);
    Business.add(Cptr);
    Business.add(Dptr);
    Business.add(CopyCptr);
    forICursor (myCursor1) {
        cout << *Business.elementAt(myCursor1);
    }
}
```

CopyCptr and Cptr refer to different memory addresses, so both of them could be put into the set. Using element pointers rather than regular pointers, all collection functions are done on the elements to which the pointers point. That is why a pointer pointing on Sandra Summers is only entered once into the list.

Element Functions That Work on the Pointers Instead of the Referenced Elements

If you do want element functions to work on the pointers instead of the referenced elements, you do not need to implement equality and ordering relation for the chosen pointer type (IAutoPointer, IMngPointer or C++ pointers). The compiler can instantiate the default element function templates in such cases. If necessary, you can implement your element functions for the referenced element type.

In the following example, adding, locating, and other functions are based on pointer equality and ordering, and not on the equality defined for the Person type.

```
//main.cpp - main file
#include "person.h" //person.h from the previous example
#include <istdops.h>
#include <iset.h>
typedef IMngPointer <Person> ManagedPersonPtr;
typedef ISet <ManagedPersonPtr> AddressList;
ostream& operator<<(ostream& os,Person A)
{
    return (os << endl << A.GetPersonName()
        << " " << A.GetTNumber());
}
void main()
{
```

```
    AddressList Business;
    AddressList::Cursor myCursor1(Business);
    ManagedPersonPtr ptrA(new Person("Peter Black",
        "714-50706"), IINIT);
    ManagedPersonPtr ptrB(new Person("Carl Render",
        "714-540321"), IINIT);
    ManagedPersonPtr ptrC(new Person("Sandra Summers",
        "x"), IINIT);
    ManagedPersonPtr ptrD(new Person("Mike Summers",
        "x"), IINIT);
    ManagedPersonPtr copyPtrC(new Person("Sandra Summers",
        "x"), IINIT);
    Business.add(ptrA);
    Business.add(ptrB);
    Business.add(ptrC);
    Business.add(ptrD);
    Business.add(copyPtrC);
    forICursor (myCursor1) {
        cout << *Business.elementAt(myCursor1);
    }
}
```

The variables copyPtrC and ptrC refer to different memory addresses, so both of them are entered into the set even if the element they point to is identical. This is because equality now refers to the pointers even though it is also defined for Person.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Smart Pointers

RELATED TASKS

Implement Element- and Key-Type Functionality
"Choose the Appropriate Smart Pointer Class" on page 55
"Construct Smart Pointers" on page 54Use Automatic Pointers
"Use Managed Pointers"

Use Managed Pointers

Managed pointers keep a reference count for each referenced object (element). When the last managed pointer to the object is destructed, the object is automatically deleted. You should use managed pointers when you are unsure who is responsible for deleting an object. This may occur where several pointers to an object are introduced over time, and the order in which the pointers are released is not known.

The following example shows how to use pointers from the IMngElemPointer class:

```
//main.cpp - main file
#include "person.h" //person.h from the previous examples
#include <istdops.h>
#include <iset.h>
typedef IMngElemPointer <PersonPtr> MEPersonPtr;
typedef ISet <MEPersonPtr> AddressList;
ostream& operator<<(ostream& os,Person A)
{
    return (os << endl << A.GetPersonName() << " " <<
        A.GetTNumber());
}
void main()
```

```
{
    AddressList Business;
    AddressList::Cursor myCursor1(Business);
    MEPersonPtr Aptr (new Person("Peter Black","714-50706"),IINIT);
    MEPersonPtr Bptr (new Person("Carl Render","714-540321"),IINIT);
    MEPersonPtr Cptr (new Person("Sandra Summers","x"),IINIT);
    MEPersonPtr Dptr (new Person("Mike Summers","x"),IINIT);
    MEPersonPtr CopyCptr (new Person("Sandra Summers","x"),IINIT);
    Business.add(Aptr);
    Business.add(Bptr);
    Business.add(Cptr);
    Business.add(Dptr);
    Business.add(CopyCptr);
    forICursor (myCursor1) {
        cout << *Business.elementAt(myCursor1);
    }
    Business.remove(Cptr); //Remove pointer from collection
}
```

After removing the pointer from the collection, the managed pointer is automatically deleted. In the example, the allocated Person will automatically be deleted by the remove function unless it is referenced through another PersonPtr.

Managed Pointers and Copying Elements
If you want to create managed pointers for a collection and copy in elements from a second collection that already contains managed pointers, you cannot use IINIT because it will destroy the managed pointers in the second collection. To avoid this situation, you can use the following notation:

```
typedef IMngElemPointer<PersonPtr> MyClassPtr;
typedef IKeySet<MyClassPtr> MyAddressList;
MyClassPtr pMyClass;
pMyClass = Business.elementWithKey(...);
```

In the above notation, `Business` is the collection from the previous examples, but here it is an IKeySet collection rather than an ISet collection so that elementWithKey can be used.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Smart Pointers

RELATED TASKS

"Choose the Appropriate Smart Pointer Class" on page 55
"Construct Smart Pointers" on page 54Use Automatic Pointers
"Use Element Pointers" on page 58

Chapter 8. Thread Safety and the Collection Classes

Like most of the IBM Open Class classes, the collection classes require thread safe operation of multithreaded access to global data. The collections may be used in a multithreaded environment, but any single instance of a collection may be referenced by only one thread at a time. Different collections may be accessed concurrently. For example, thread 1 references collection A concurrently with thread 2 referencing collection B.

The collection classes offer built-in Guard objects to simplify the explicit serialization needed to protect the collection instance. While serialization for global data is still necessary, the built-in support helps to reduce the amount of programming required. Note, however, that the locking of elements stored within a collection is the responsibility of the user and is not provided as part of collection class thread safety.

Restrictions
The current implementation does not provide any means to support users who want to program in a multiprocessing environment with the Collection Classes. These classes provide no built-in serialization support for multiple processes. As a result, you cannot share collection objects between multiple processes without adding your own serialization mechanism. The Collection Classes only support thread safety within a single process.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Open Class Threading Model

RELATED TASKS

Insure Thread Safety with Guard Objects
"Instantiate a Guard Object" on page 64
"Use Guard Objects" on page 65
Start a Thread

Insure Thread Safety with Guard Objects

For each different collection abstraction, a Guard class similar to IResourceLock has been defined and a corresponding typedef added:

```
template <class Element> class ICollectionGuard { ... };
typedef ICollectionGuard<Element> guard;
```

Essentially, a Guard object is an object created on a stack that is used to lock some other object. Guard objects are useful in C++ because they respond properly to exceptions. When an exception is thrown while still in the scope of the Guard object, its destructor is called as the exception passes through the stack frame and the destructor unlocks the target object. As a result, the exception can be caught and dealt with by code further up the call chain without leaving the locked object in an unusable locked state.

The Guard typedef can be used as if it was a nested class of a particular collection:

```
template <class Element>
class ICollectionGuard
{
public:
   ICollectionGuard(
      IACollection<Element>&, long timeout = -1);
      ~ICollectionGuard();
private:
    IACollection<Element>& ivCollection;
};
```

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
"Chapter 8. Thread Safety and the Collection Classes" on page 63
Open Class Threading Model

RELATED TASKS

"Instantiate a Guard Object"
"Use Guard Objects" on page 65

Instantiate a Guard Object

ICollectionGuard<Element> Constructor and Destructor
The Guard constructor takes the collection object to be locked and an optional timeout value as parameters. The timeout value is specified in milliseconds. If a lock request cannot be resolved within the specified range of time, an exception is thrown. The timeout value defaults to -1 to indicate an indefinite wait. The value 0 informs the constructor to throw an exception if the lock is not immediately available.

WIN

This parameter is only supported on non-POSIX (Windows) platforms. Other platform implementations ignore the specification of this value.

The Guard destructor unlocks the Collection specified within the constructor of the Guard.

Guard Copy Constructor
The Guard copy constructor is made private in order to prevent the user from copying Guard objects.

Collection Constructor and Destructor
The collection does not keep track of all possible Guard objects currently in use with the target collection. Guards for a collection must be destructed before the collection itself is destructed. This is normally accomplished by declaring the Guard within a compound statement so that it is automatically destructed when the statement passes out of scope.

Collection Copy Constructor
If a new collection is created from an existing collection instance, the guards of the existing collection have no effect on the new collection.

Return Codes and Exceptions
Since the Guard is constructed, there are no return codes. The Collection classes use exceptions to indicate that a lock cannot be obtained. The user must code the

Guard constructor within a try/catch clause. When the Guard constructor fails and the lock was not obtained for any reason, a C++ exception is thrown.

Deadlocks
In either of the above cases, you are responsible for the proper sequence of obtaining the locks. There is no special code within the collection classes to prevent the user from producing deadlocks.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
"Chapter 8. Thread Safety and the Collection Classes" on page 63
Open Class Threading Model

RELATED TASKS

Insure Thread Safety with Guard Objects
"Use Guard Objects"

Use Guard Objects

In a user program, a Guard is used in the following way to obtain a lock on a specific collection:

```
ISet<char> my_set;
try
{
   ISet<char>::Guard g(my_set);
   my_set.add('x');
}
catch (IException& e)
{
   // handle exception
}
```

The critical region, in this case the add method invoked on the ISet<char>, must be specified within a C++ compound statement. On entry to the block, the Guard constructor locks the collection that is specified as the Guard constructor parameter. The destructor is executed when the scope of the block is left at the time the collection is unlocked. The specified name of the Guard object (g in the above example) is arbitrary and plays no role in the locking.

Depending on the number of threads of a particular user application, multiple Guard objects may exist that work with the same collection object.

For the Restricted Access Collections and the Tree Collections, two similar Guard classes and corresponding typedefs are added. They are exposed to the user through the following typedefs on the level of appropriate concrete collections:

```
typedef IRestrictedAccessCollectionGuard<Element> guard;
typedef ITreeGuard<Element> guard;
```

In the event that the user invokes a Collection method that involves two or even three collections, code such as the following must be used in order to achieve thread-safe execution:

```
try
{
   ISet<char>::Guard l1(my_set1);
   ISet<char>::Guard l2(my_set2);
   my_set1.addAllFrom(my_set2);
}
```

```
catch (IException& e)
{
   // handle exception
}
```

In the case of three involved collections, the following code must be used:

```
try
{
   ISet<char>::Guard l1(my_set1);
   ISet<char>::Guard l2(my_set2);
   ISet<char>::Guard l3(my_set3);
   my_set1.addInterSection(my_set2,my_set3);
}
catch (IException& e)
{
   // handle exception
}
```

In cases such as these, where multiple locks must be acquired, it is important that each section of code that acquires the locks do so in the same order. Not doing so can result in deadlocks.

The programmer does not need to include any new header files. The typedef for the ISet coding samples illustrated above is provided by the standard include file iset.h.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
"Chapter 8. Thread Safety and the Collection Classes" on page 63
Open Class Threading Model

RELATED TASKS

Insure Thread Safety with Guard Objects
"Instantiate a Guard Object" on page 64

Chapter 9. Support for Notifications

The Collection Classes include special classes that support notifications. For every concrete flat collection class (for example ISequence), there is a corresponding notification-enabled collection class (for example IVSequence).

All collection methods that modify a collection send notifications to observers. The class IVCollection defines four notification IDs for collection classes:

Notification ID	Description
addId	Sent if an element is added to the collection.
removeId	Sent if an element is removed from the collection.
replaceId	Sent if an element is replaced in the collection.
modifyId	Sent if a collection is changed in any way other than those mentioned above.

For notifications addId, removeId and replaceId, you can use INotificationEvent::eventData to access event data generated by collections. This event data is an object that includes a cursor method to access a collection cursor. The cursor points to the element referred to by the modification method. For example, if addId is the notification, the cursor points to the added element. The replaceId notification also gives you access to a copy of the element that was replaced.

Collection notifications addId, removeId and replaceId pass a pointer to the class IVCollectionEventData.

For the notifications addId and modifyId, the library sends notification after the modification occurs. For the notification removedId and replaceId the library sends notification before the collection is changed, otherwise you would not be able to use the cursor to refer to the element being removed.

Notifications are only sent if the collection is changed by the method. The following methods do not create a notification:

- removeAll() for an empty collection
- add(), when add() does not actually add an element (for example, because the element already exists in a unique collection, or because the collection is full)
- remove() if the element is not in the collection
- locateOrAdd() if the element is already in the collection

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
"Chapter 3. Addition, Removal, and Replacement of Elements" on page 27
Event Notification

RELATED TASKS

"Use Collection Notification"
"Remove an Element from a Collection" on page 29Notify Observers Synchronously

Use Collection Notification

The following example demonstrates the use of collection event data for a sequence of IString objects. IString is the main string handling class provided by the IBM Open Class Library.

```
#include <iobservr.hpp>
#include <inotifev.hpp>
#include <iseq.h>
#include <iostream>
template <class Notifier>
class Observer : public IObserver
{
public:
    Observer(Notifier* notifier)
            : ivNotifier(notifier)
    {
        handleNotificationsFor(*ivNotifier);
    }
     Observer()
    {
        if (ivNotifier != 0)
            stopHandlingNotificationsFor(*ivNotifier);
    }
    IObserver&
    dispatchNotificationEvent(INotificationEvent const& event)
    {
        if (event.notificationId() == IVCollection::removeId)
        {
            std::cout << "IVCollection::removeId received" << std::endl;
        }
        else if (event.notificationId() == IVCollection::replaceId)
        {
            std::cout << "IVCollection::replaceId received" << std::endl;
        }
        else if (event.notificationId() == IVCollection::addId)
        {
            std::cout << "IVCollection::addId received" << std::endl;
        }
        else {
            std::cout << "Unknown event received" << std::endl;
        }
        return *this;
    }
private:
    Notifier* ivNotifier;
};
int
main(int argc, char **argv)
{
    IVSequence<long> seq;
    IVSequence<long>::Cursor c(seq);
    Observer<IVSequence<long> > observer(&seq);
    seq.enableNotification();
    seq.add(123,c);
    std::cout << "element in collection: "
              << seq.elementAt(c) << std::endl;
    seq.replaceAt(c,456);
    std::cout << "element in collection: "
              << seq.elementAt(c) << std::endl;
    seq.removeAt(c);
```

```
    std::cout << "Number of elements in collection: "
              << seq.numberOfElements() << std::endl;
    return 0;
}
```

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
"Chapter 9. Support for Notifications" on page 67
Event Notification

RELATED TASKS

Notify Observers Synchronously

Chapter 10. Collection Class Library Exceptions

Exceptions Defined and Used by the Collection Classes
The following is a hierarchy of the exceptions defined and used in the Collection Classes. They all derive from IException, directly or indirectly.

- IException
  - ICollectionResourceException
  - ICollectionLockException
  - ICollectionUnlockException
  - ICollectionLockTimeOutException*
  - IPreconditionViolation
    - IChildAlreadyExistsException
    - ICursorInvalidException
    - ICyclicAttachException
    - IEmptyException
    - IFullException
    - IIdenticalCollectionException
    - IInvalidReplacementException
    - IKeyAlreadyExistsException
    - INotBoundedException
    - INotContainsKeyException
    - IPositionInvalidException
    - IRootAlreadyExistsException
  - IResourceExhausted
    - IOutOfMemory
      - IOutOfCollectionMemory

* AIX This exception is not available on AIX. Due to inherent POSIX limitations, the IBM Open Class collection classes do not support *time-out processing* in the AIX environment. (Time-out processing allows you to specify a time limit for acquiring a resource lock.)

Exception Causes
A *precondition* of a called function is a condition that the function requires to be true when it is called. The calling function must assure that this condition holds. The called function implementation may assume that the condition holds without further checking it. If a precondition does not hold, the called function's behavior is undefined.

If you want to make your programs more robust and to locate errors in the test phase, the functions your program calls should check to ensure that their preconditions hold. The Collection Class Library enables this checking through macro definitions. Because this checking often requires significant overhead, it is turned off by default. You need only use it while you are testing the system and verifying that preconditions are always met. See "Enable Exception Checking" on page 74 in the Tasks section to learn how to turn this checking on and off.

A call to a function that violates the function's preconditions has two possible results:

- If the called function checks its preconditions, the function will throw an exception.
- If the function does not check its preconditions, the behavior of the function is undefined.

Precondition-Violations
The Collection Classes include the following precondition-violation exceptions:

Exception	Description
IChildAlreadyExistsException	Occurs when you try to add a child to a tree using addAsChild at a position that already contains a child.
ICursorInvalidException	Two cursor properties may lead to the ICursorInvalidException: 1. Every time a cursor is created, you must specify the collection that it belongs to. If a function takes a cursor as an argument (such as add, setToFirst, and locate), the function can only be applied to the collection that the cursor belongs to. If the function is applied to another collection, the ICursorInvalidException results. 2. If a function takes a cursor as an input argument (such as elementAt, removeAt, and replaceAt), the cursor must be valid. A cursor is valid if it actually refers to some element contained in the collection. You can use the isValid function to determine if a cursor is valid.
ICyclicAttachException	Occurs when a function tries to attach a child or a subtree at a position of a tree (using the functions attachAsChild or attachSubtreeAsChild, respectively) while that child or subtree is the tree itself.
IEmptyException	Occurs when a function tries to access an element of an empty collection. Functions that might cause this exception include firstElement and removeFirstElement.
IFullException	Occurs when a function tries to add an element to a bounded collection that is already full. Functions that might cause this exception include add and addAsFirst.
IIdenticalCollectionException	Occurs when the function addAllFrom is called with the source collection being the same as the target collection.
IInvalidReplacementException	Occurs when, during a replaceAt function, the replacing element has different positioning properties than the positioning properties of the element to be replaced.

Exception	Description
IKeyAlreadyExistsException	Occurs when a function attempts to add an element to a map or sorted map that already has a different element with the same key. Functions that might cause this exception include add and addAllFrom.
INotBoundedException	Occurs when the function maxNumberOfElements is applied to a collection that is not bounded.
INotContainsKeyException	Occurs when the function elementWithKey is applied to a collection that does not contain an element with the specified key.

Other Exceptions
The following lists the causes of the other exceptions defined or used by the Collection Classes:

Exception	Description
IOutOfCollectionMemory	Occurs when the collection classes cannot allocate any memory for its data structures, including pointers and cursors. This class inherits from IOutOfMemory.
IPositionInvalidException	Occurs when a function specifies a position that is not valid in a collection. The functions that might cause this exception include elementAtPosition, removeAtPosition, and setToPosition.
IRootAlreadyExistsException	Occurs when the function addAsRoot is called for a tree that already has a root.
ICollectionResourceException	Occurs when the Collection is constructed and the creation of the internal Resource object fails.
ICollectionLockException	Occurs when an internal lock request fails.
ICollectionUnlockException	Occurs when an internal unlock request fails.
ICollectionLockTimeOutException	Occurs when a time-out value is specified during the construction of a Guard object and the lock cannot be obtained within the specified period of time.

Exceptions Caused by System Failures and Restrictions
System failures and restrictions are different from precondition violations. You cannot usually anticipate them, and you have no opportunity to verify that such situations, for example storage overflow, will not occur. These exceptions need to be checked for, and an exception should be thrown if they occur.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Exceptions in the IBM Open Class

RELATED TASKS

"Enable Exception Checking"
"Chapter 11. Troubleshoot Collection Class Problems" on page 77

Enable Exception Checking

Some preconditions are more difficult to check than others. Consider the following possible preconditions:

1. A cursor for a linked collection implementation still points to an element of a given collection.
2. A collection is not empty.

In the production version of a program, it may be less efficient to check the first precondition than the second.

The Collection Class Library provides three levels of precondition checking. They are selected by the following C-preprocessor macro definitions:

Macro	Description
Default	Perform all precondition checks, except the check that a cursor actually points to an element of the collection.
INO_CHECKS	Check for memory overflow. Other checks may be eliminated to improve performance.
IALL_CHECKS	Perform all precondition checks, including the (costly) check that a cursor actually points to an element of the collection. This extra check can only fail for undefined cursors.

Define the C-preprocessor macro to use them. You typically define them with the appropriate compiler options. For example, you could include the following directive in your project's configuration file:

```
define("INO_CHECKS")
```

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
"Chapter 10. Collection Class Library Exceptions" on page 71
Exceptions in the IBM Open Class

RELATED TASKS

"Enable Exception Checking"
"Handle Exceptions with Cursors"
"Chapter 11. Troubleshoot Collection Class Problems" on page 77

Handle Exceptions with Cursors

Exceptions are not generally used to change the flow of control of a program under normal circumstances. An example of using exceptions under normal circumstances is a function that iterates through a collection, and exits from the iteration by checking for the exception that is thrown when an invalid cursor is used to access elements. When the iteration is complete, the cursor will no longer be valid, and this exception will be thrown. This is not a good programming

practice. A function should explicitly test for the cursor being valid. To make this possible, a function must efficiently test this condition (isValid, for the cursor example).

There are situations where the test for a condition can be done more efficiently in combination with performing the actual function. In such cases, it is appropriate, for performance reasons, to make the situation regular (that is, not exceptional) and return the condition as a IBoolean result.

Consider a function that first tests whether an element exists with a given key, and then accesses it if it exits:

```
if (c.containsElementWithKey (key)) {
    // ...
    myElement = c.elementWithKey (key); // inefficient
    // ...
}
else {
    // ...
}
```

This solution is inefficient because the element is located twice, once to determine if it is in the collection and once to access it. Consider the following example:

```
try {
    // ...
    myElement = c.elementWithKey (key); // bad: exception expected
    // ...
}
catch (INotContainsKeyException) {
    // ...
}
```

This solution is undesirable because an exception is used to change the flow of control of the program. The correct solution is to obtain a cursor together with the containment test, and then to use the cursor for a fast element access:

```
if (c.locateElementWithKey (key, cursor)) {
    // ...
    myElement = c.elementAt (cursor); // most efficient
    // ...
}
else {
    //...
}
```

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
"Chapter 10. Collection Class Library Exceptions" on page 71
Exceptions in the IBM Open Class
"Chapter 6. Iteration" on page 47

RELATED TASKS

"Enable Exception Checking" on page 74
"Handle Exceptions with Cursors" on page 74
"Chapter 11. Troubleshoot Collection Class Problems" on page 77
"Use Cursors to Locate and Access Elements" on page 48
"Use Cursors to Iterate Over a Collection" on page 50
"Use allElementsDo and Applicators to Iterate Over a Collection" on page 51

Chapter 11. Troubleshoot Collection Class Problems

The following table provides a short summary of problems you may encounter when you use the Collection Class Library, and directs you to a section containing hints for a solution.

Problem effect	Problem area
"Compilation Errors Indicating a Problem with Constructors"	Default constructor is missing
"Compilation Errors Indicating that an Element Type or Function is not Declared" on page 78	Declaration of template arguments and element functions not recognized
"Compilation Errors about Multiple Definitions" on page 78	Header files may be included more than once
"Compiler Warning of an Error in istdops.h" on page 79	Global key function return value incorrect
"Link or Bind Errors about Multiple Definitions" on page 80	Header files may be included more than once
"Link or Bind Error Indicating istdops.h" on page 80	No declaration of global key function
"Unexpected Exception Tracing Output" on page 81	Trace function write is called whether or not the related exception is caught
"Unexpected Results when Adding an Element to a Unique Key Collection" on page 82	Global key function return value incorrect
"Unexpected Results when Using Cursors" on page 82	Undefined cursor used

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1

Compilation Errors Indicating a Problem with Constructors

Effect

You get a compiler error about a constructor. The errors you encounter may not be exactly the same as the following:

OS/2

The following is the message text from OS/2®:

```
itbseq.h(25:1) : (E) EDC3222:
"IGTabularSequence<ToyString,IStdOps<ToyString> >::Node" needs a
constructor because class member "ivElement" needs a constructor
initializer.
Names namesOfExtinct(animals.numberOfDifferentKeys());
animals.cpp(55:57) : informational EDC3207:
The previous message applies to the definition of template
"ITabularSequence<ToyString>".
```

AIX

The following is the message text from AIX:

```
"/usr/vacpp/include/itbseq.h", line 25.1: 1540-222:
  (S) "IGTabularSequence<ToyString,IStdOps<ToyString> >::Node"
  needs a constructor because class member "ivElement" needs a
  constructor initializer.
"animals.cpp", line 55.1: 1540-207:
  (I) The previous message applies to the definition of template
  "ITabularSequence<ToyString>".
```

Reason
Compiler error messages indicating a problem with constructors for a collection are typically related to the constructors defined for your element. Here the default constructor for the element is missing.

Solution
Define the default constructor for the element class.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes

RELATED TASKS

"Chapter 11. Troubleshoot Collection Class Problems" on page 77

Compilation Errors Indicating that an Element Type or Function is not Declared

Effect
You get compiler messages when processing templates indicating that an element type or one of its required element functions is not declared.

Reason
The element type or element function is defined locally to the source file that contains the template instantiation with the element type as its argument. The prelink or prebind phase is executed only by using the header files. Therefore, your declaration local to a source file is not recognized and causes these compilation errors.

Solution
Move the corresponding declarations to a separate header file and include the header file from the source file.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes

RELATED TASKS

"Chapter 11. Troubleshoot Collection Class Problems" on page 77

Compilation Errors about Multiple Definitions

Effect
You get compilation errors from symbols being defined multiple times.

Reason
The template instantiation needs to include the type declarations it received as arguments. Your header files containing type declarations used in template classes may automatically be included several times.

Solution
Protect your header files against multiple inclusion by using the following preprocessor macros at the beginning and end of your header files:

```
#ifndef _MYHEADER_H_
#define _MYHEADER_H_ 1
...
#endif
```

Where _MYHEADER_H_ is a string, unique to each header file, representing the header file's name.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes

RELATED TASKS

"Chapter 11. Troubleshoot Collection Class Problems" on page 77

Compiler Warning of an Error in istdops.h

Effect
You get a compiler warning indicating a problem in istdops.h (the errors you encounter may not be exactly the same as the following):

OS/2

The following is the message text from OS/2:

```
j:\...\include\istdops.h(166:1) : warning EDC3285:
  The address of a local variable or compiler temporary is being used
  in a return expression.
j:\...\include\istdops.h(160:1) : informational EDC3207:
  The previous message applies to the definition of template
  "IStdKeyOps<Word,int>::key(const Word&) const".
```

AIX

The following is the message text from AIX:

```
"/usr/vacpp/include/istdops.h", line 166.1: 1540-151:
  (I) auto compiler temporary of type "int" has been generated.
"/usr/vacpp/include/istdops.h", line 160.1: 1540-207:
  (I) The previous message applies to the definition of template
  "IStdKeyOps<Word,int>::key(const Word&) const".
```

Reason
Compiler error messages indicating a problem in istdops.h are related to the element and key-type functions that you must define for your elements. These functions depend on the collection and implementation variant you are using.

Your global-name-space function key returns the key by value instead of by reference. A temporary variable is created for the key within the operator-class

function key. The operator class function key returns the key by reference. Returning a reference to a temporary variable causes unpredictable results.

The key function must return a reference and must also take a reference argument. If the key function calls other functions to access the key, it must call those functions with a reference to the object as an argument, and those functions must return a reference to the key.

Solution
Verify that the global name-space function key correctly returns a key const& instead of key.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes

RELATED TASKS

"Chapter 11. Troubleshoot Collection Class Problems" on page 77

Link or Bind Errors about Multiple Definitions

Effect
You get link or bind errors from symbols being defined multiple times.

Reason
The template instantiation needs to include the type declarations it received as arguments. Your header files containing type declarations used in template classes might automatically be included several times.

Solution
Verify that you did not define functions in the header files that declare types used in templates. If you did, you must move them from the header file into a separate source file or make them inline.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes

RELATED TASKS

"Chapter 11. Troubleshoot Collection Class Problems" on page 77

Link or Bind Error Indicating istdops.h

Effect
You are using a collection class with a key, and you get an error message during the link or bind step indicating a problem in istdops.h (**Note**: the errors you encounter may not be exactly the same as the following):

OS/2

The message text for OS/2 is:

```
istdops.h(176): (E) EDC3013: "key" function is undefined.
```

AIX

The message text for AIX is:

```
":/usr/lpp/xlC/include/istdops.h", line 176:
1540-3013:
  (S) "key" is undefined.
```

AIX

The message text for Solaris is:

```
"/.../.../include/istdops.h", line 176:
  (S) "key" is undefined.
```

Reason
You are using a collection class that requires the element class to provide a key and you chose to use the method of using a global key function. You are using collection class methods in a source file but the header file with the same name as the source file does not contain a declaration (prototype) of the global key function.

While compiling the source file, which uses methods of the collection class, the compiler has created or modified a temporary source file in the tempinc directory. During the link or bind step, bind step, this source file is compiled to resolve references to template code. The error message you encounter refers to this compilation. The source file in the tempinc directory contains include directives for the collection class template code. It also contains include directives for a header file of the same name as the source file that uses the collection class methods. The template code in istdops.h requires that the global key function be known at compilation time. The only file that is included at this time is the header file with the same name as your source file. The problem is that the source file is not included at this time, so a definition or declaration of the global key function in this file is not recognized by the compiler.

Solution
You must declare the global key function in the header file with the same name as the source file that uses the collection class methods. The definition of the global key function should be in the source file. If you are not sure which header file is meant by the message, look in the source file found in the tempinc directory.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes

RELATED TASKS

"Chapter 11. Troubleshoot Collection Class Problems" on page 77

Unexpected Exception Tracing Output

Effect
You get unexpected exception tracing output on standard error, even though the related exception causing the output is caught.

Reason
For each exception raised, the write function of class IException::TraceFn is called and writes information about the raised exception to standard error. This trace function write is called whether the related exception is caught or not.

Solution
To suppress the trace output, provide your own IException::TraceFn::write tracing function by subclassing IException::TraceFn and register the subclass with setTraceFunction.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes
Exceptions in the IBM Open Class

RELATED TASKS

"Chapter 11. Troubleshoot Collection Class Problems" on page 77
Trace Exceptions

Unexpected Results when Adding an Element to a Unique Key Collection

Effect
You are adding an element into a unique key collection, such as a key set or a map, and you are sure that the collection does not yet contain an element with the same key. Nevertheless, you get unexpected results: IKeyAlreadyExistsException, or the element is not added and the cursor is positioned to a different element.

Reason
This problem has the same cause as the problem described for "Compiler Warning of an Error in istdops.h" on page 79. However, you did not get the warning message described above, because you compiled with a lower warning level.

Solution
Verify that the global key function correctly returns a key const& instead of key.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes
"Chapter 3. Addition, Removal, and Replacement of Elements" on page 27

RELATED TASKS

"Chapter 11. Troubleshoot Collection Class Problems" on page 77
"Add an Element to a Collection" on page 29

Unexpected Results when Using Cursors

Effect
You get unexpected results when using cursors. For example, the elementAt function fails for the given cursor or returns an unexpected element.

Reason
You have used an undefined cursor. Cursors become undefined when an element is added to or removed from the collection.

Solution
Cursors that become undefined must be rebuilt with an appropriate operation (for example, locate) before they are used again. Rebuilding is especially important for

removing all elements with a given property from a collection. Elements cannot be removed by coding a cursor iteration. Use the removeAll function that takes a predicate function as its argument.

RELATED CONCEPTS

"Chapter 1. Collection Classes Overview" on page 1
Hierarchy and Design of the Collection Classes
"Chapter 6. Iteration" on page 47

RELATED TASKS

"Chapter 11. Troubleshoot Collection Class Problems" on page 77
"Use Cursors to Locate and Access Elements" on page 48
"Use Cursors to Iterate Over a Collection" on page 50