VisualAge® C++ Professional for AIX®

IBM® Open Class™: Text and Internationalization

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. International Framework

The International Framework provides classes for creating international applications and manipulating international text. International applications are those that can be localized for different languages and geographic regions without accessing the source code. International text may consist of translated text strings as well as different character sets.

Locales
A locale is typically defined by a combination of language and geographical region, but other elements, such as date, time, currency, and number formats, may also help to form the locale definition.

The locale mechanism provides access for localizable resources. Localizable resources include any objects—such as transcoders, collation objects, number formatters, date formatters, and time zones—that might change to support different languages or geographic regions. For example, collation ordering rules for text sorting are associated with particular languages, while date, time, and currency formats are more closely associated with particular regions.

ILocaleKey lets you access both host-specific and portable resources associated with a particular locale. Neither ILocaleKey nor the particular locale in question owns the objects associated with it.

Collation
Collation classes enable both language-sensitive and language-insensitive string comparison. For language-sensitive comparison of Unicode text strings, the collation classes use the alphabetical ordering rules of a natural language instead of the numeric values of the character encoding system.

Transcoding
Using a set of mapping rules to convert text data between two character encoding standards is called transcoding. The Open Class transcoding classes enable conversion of Unicode character data to and from other character encoding systems, including ASCII and the code pages used in Microsoft® environments. The transcoding classes also handle the conversion of special characters.

The LOCPATH environment variable identifies the location of international resources on your system. You must define this environment vairable so that the C runtime can find the locale data files.

RELATED CONCEPTS

"Locales in Internationalization" on page 2
"Locale Classes" on page 3
"Collation Classes" on page 8
Transcoding Classes
"Transcoder Names" on page 20

Locales in Internationalization

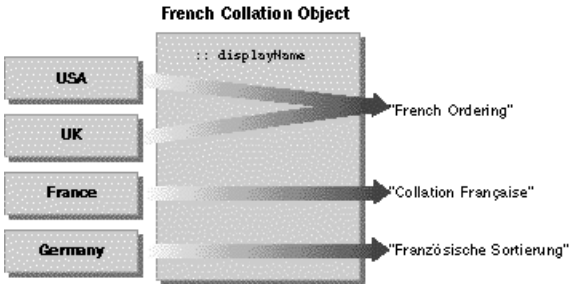
Locales provide the mechanism for organizing and accessing localizable resources—any objects that might change to support different languages or geographic regions. Such objects can include transcoders, collation objects, number formatters, data formatters, and time zones, among others.

Object Portability
Localized objects fall into one of two categories:

- *Host* objects are specific to a particular platform implementation. You cannot typically build or modify them programmatically, nor can you stream them to other systems. They are usually tied to a specific locale, and you construct them using the key for that locale.
- *Portable* objects are not host-dependent. You can typically build them programmatically from data, edit them, and stream them to other systems. You can also construct them using a locale key if they have been associated with that locale. Portable objects can be shared between locales, but they are not required to be associated with any locale.

No locale owns the objects associated with it; the locale simply provides a way to access those objects.

Names of International Objects
The locale mechanism allows each locale and each object associated with that locale to have a set of displayable names. At a minimum, each object has a name for its locale in the default language and in English. Objects can also have names in other languages for display to end users, as illustrated in this figure:



RELATED CONCEPTS

"Chapter 1. International Framework" on page 1
"Locale Classes" on page 3
"Collation Classes" on page 8
Transcoding Classes
"Transcoder Names" on page 20
"Date and Time Classes" on page 27
"DBCS and National Language Support" on page 40

RELATED TASKS

"Get a Key for a Locale" on page 6
"Get an Object from the Current Default Locale" on page 6
"Get an Object from a Specific Locale" on page 7
"Iterate through Available Locales" on page 7

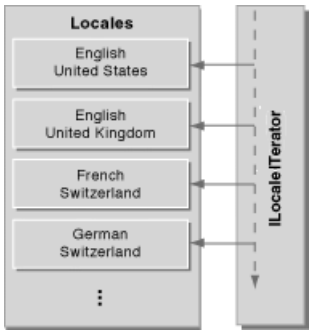
Locale Classes

The locale classes consist of ILocaleKey, which gives you access to the resources associated with a particular locale, and ILocaleKeyIterator, which lets you iterate through the locally installed locales.

VisualAge C++ provides national language support using the XPG/4 programming model, and using the locale-sensitive functions of the C runtime library.

The IBM Open Class uses the C runtime locale across all platforms.

This figure illustrates how locales can be grouped by language (as for English) or by geographical location (as for Switzerland):



Access to Locale Resources
The primary class you use to work with locales is ILocaleKey. ILocaleKey acts as an identifier for a specific locale and is used as a parameter for functions that create a particular type of localizable object. For example, the ICollation::createCollation function takes an ILocaleKey object that indicates what language the collation should reflect. On the other hand, the ILocaleKey::displayName function lets you display the locale name in any language for which a name is provided. Each type of international object should provide a displayName function, such as ICollation::displayName.

When you need a localized object, create an ILocaleKey object to define the locale with which to associate the object. A given object can be associated with more than one locale. Thus, a collation object for English could be associated with many, or even all, English-speaking locales (United States, Canada, United Kingdom, Jamaica, and so on). The set of objects associated with a locale can be open-ended; you are free to associate new objects with the locale.

Use ILocaleKey to identify the locale for which you want to access localized resources. You can use ILocaleKey to specify a particular locale (such as U.S. English) or to specify the default locale on the current host. ILocaleKey gives you access to both host-specific and portable objects associated with a given locale.

ILocaleKey supports two ways to describe a specific locale:

- Using the host locale ID (an unsigned long value) for the current platform. For example, on Win32 platforms the locale identifier for United States English is 0x0409.
- Using the portable POSIX identifier. This has three parts:
  - A two-character language identifier, for example, “EN” for English or “ZH” for Chinese.

- A two-character region identifier, for example, “US” for the United States or “TW” for Taiwan.
- A single-character extension used in a few cases where the standard “language_region” POSIX identifier maps to multiple, distinctive locales. For example, the “NO_NO” identifier maps to both the BokmÔl Norwegian and Nynorsk Norwegian locales. An extension lets you provide a unique identifier for each.

When you use a locale key to obtain an object for a given locale, the system does the following:

1. If there is a corresponding host object for that locale, the system returns it.
2. If there is no corresponding host object for that locale, the system looks for an object that corresponds to the same language.
3. If there are no corresponding host objects, the system looks for a corresponding portable object.
4. If there is no corresponding portable object, the system throws an exception.

The following chart describes the interface for ILocaleKey:

Member function	**Description**
languageID and setLanguageID	Let you access the language identifier portion of the locale ID. Two special language identifiers, “C” and “POSIX,” indicate the POSIX C locale. If you specify one of these language identifiers, the region and extension are set to null. If you set the language identifier to a null string, the locale key is set to the default host locale.
regionID and setRegionID	Let you access the region identifier portion of the locale ID.
extension and setExtension	Let you access the extension used to uniquely identify locales whose POSIX IDs map to more than one locale.
posixID and setPosixID	Let you access the entire POSIX identifier string. This string has the format “fLanguageID_fRegionIDfExtension,” for example, “ES_ESM.”
hostID and setHostID	Let you access the host identifier. These functions allow you to create a locale object using Windows LCID identifiers.
isCLocale	Indicates whether the locale key references the C (POSIX) locale.
defaultLocale	A static function that returns a locale key for the current default locale.

Member function	Description
displayName	Returns a localized, displayable name for the locale. You specify the locale in which to display the name and a lookup strategy for finding the name: • kExactLocale: Return the display name only for the exact locale specified. • kExactLanguage: If there is no name for the exact locale specified, return the name from another locale with the same language identifier. • kAnyLanguage: If there is no name in the specified locale or language, return the name in the default locale and language. This is the default option.

Locale Iteration
Use ILocaleKeyIterator to iterate through the locales available on the current hosts. Because ILocaleKeyIterator iterates through currently installed locales (rather than all locales supported by the locale mechanism), it may not reflect the full set of supported locales. You do not need to initialize an ILocaleKeyIterator object; when constructed, it automatically references the first locale key in the list.

RELATED CONCEPTS

"Chapter 1. International Framework" on page 1
"Locales in Internationalization" on page 2
"Collation Classes" on page 8
Transcoding Classes
"Transcoder Names" on page 20

RELATED TASKS

"Get a Key for a Locale" on page 6
"Get an Object from the Current Default Locale" on page 6
"Get an Object from a Specific Locale" on page 7
"Iterate through Available Locales" on page 7

Locale Names

AIX

For each fileset you want to use, you must explicitly install the corresponding fileset.

RELATED CONCEPTS

"Locale Classes" on page 3
"Chapter 1. International Framework" on page 1

Set the Locale

To use national language support you must set the locale for your program, using the setlocale function:

```
setlocale(LC_ALL,"");
```

The setlocale function call should be the first call in main(), before any IString variables are defined.

RELATED CONCEPTS

"Locale Classes" on page 3

RELATED TASKS

"Get a Key for a Locale"
"Get an Object from the Current Default Locale"
"Get an Object from a Specific Locale" on page 7

Get a Key for a Locale

When you create a locale key, you generally want a key either for the current default locale or for a specific locale. To get the key for the current default locale, use the ILocaleKey::defaultLocale function. For example:

```
ILocaleKey key(ILocaleKey::defaultLocale());
```

To get a key for a specific locale, use the locale ID. Each locale has a portable ID defined by POSIX consisting of a language identifier, a region identifier, and a host-specific ID defined by the particular platform. For example, to create a locale key for the German-speaking part of Switzerland:

```
ILocaleKey *key = new ILocaleKey("DE", "CH");
```

RELATED CONCEPTS

"Locale Classes" on page 3
"Locale Names" on page 5

RELATED TASKS

"Set the Locale" on page 5
"Get an Object from the Current Default Locale"
"Get an Object from a Specific Locale" on page 7

Get an Object from the Current Default Locale

Many classes that are instantiated into localized objects provide static functions for creating the object for a particular locale. To get the object for the current default locale, you can do the following:

1. Call the localizable object's create function without specifying a locale:

   ```
   // Call the createCollation function
   ICollation *coll = ICollation::createCollation();
   ```

2. Call the static function ILocaleKey::defaultLocale to get a locale key for the current default and pass that key to the localizable object's create function:

   ```
   // Create a key for the current default locale
   ILocaleKey defaultKey(ILocaleKey::defaultLocale());
   // Call the createCollation function
   ICollation
      *coll = ICollation::createCollation(defaultKey);
   ```

RELATED CONCEPTS

"Locale Classes" on page 3
"Locale Names" on page 5

RELATED TASKS

"Set the Locale" on page 5
"Get a Key for a Locale" on page 6
"Get an Object from a Specific Locale"

Get an Object from a Specific Locale

To get an object from a specific locale, you need to know either the POSIX identifier or the host identifier for that locale. You then create a locale key from that identifier and use the key to instantiate the correct object.

If using POSIX identifiers, you need to specify either the language identifier or both the language and region identifiers. Certain locales also require an extension to distinguish them (for example, Modern and Traditional Spanish Sort).

To get a localized object from a specific locale:

1. Create an ILocaleKey for the locale and pass in the identifier for it.
2. Call the localized object class's create function and pass in the key for the specific locale.

For example, this code shows how to get the collation object for the French Canadian locale:

```
// Get the collation object for the
// French Canadian locale
try
{
   ICollation
      *coll = ICollation::createCollation
                 (ILocaleKey("FR", "CA"));
}
catch (IException&)
{
   // If it isn't available,
   // use the collation for the default
   locale coll = ICollation::createCollation();
}
```

RELATED CONCEPTS

"Locale Classes" on page 3
"Locale Names" on page 5

RELATED TASKS

"Set the Locale" on page 5
"Get a Key for a Locale" on page 6
"Get an Object from the Current Default Locale" on page 6

Iterate through Available Locales

Use ILocaleKeyIterator to iterate through the available locales. ILocaleKeyIterator iterates through the locales currently installed on the system, not the complete set of supported locales.

The following steps show how to use ILocaleKeyIterator:

1. Instantiate an ILocaleKeyIterator.
2. Use operator++ or operator— to move forward or backward through the list of installed locales.

3. Use operator* to return a key for the locale currently referenced by the iterator or use operator-> to return a constant pointer to the key.

For example, this code shows how to iterate through the available locales, printing out the English name for each locale:

```
// Create the iterator
ILocaleKeyIterator iter;
// Create a key for an English locale
ILocaleKey english("EN");
while (iter)
{
   // Get the name of the current locale
   IText name = (*iter).displayName(english);
   cout << name << "/n";
   cout.flush();
   // Increment the iterator to the next
   // locale in the list
   iter++;
}
```

RELATED CONCEPTS

"Locale Classes" on page 3
"Locale Names" on page 5

RELATED TASKS

"Iterate through Available Collation Objects" on page 15
"Iterate through Available Locales" on page 7
"Iterate through Available Transcoders" on page 26

Collation Classes

Overview of Collation Classes

In most cases, the ordering of Unicode values does not produce correct ordering results. For example, in the ASCII-based character sets, *Z* is ordered before *a*, and *z* is ordered before ±. Open Class collation classes, however, support collation objects that compare strings based not on the Unicode values of each character, but on the rules of a natural language. This is what enables language-sensitive string comparison.

Each International Framework collation object is based on a set of rules that define the results for alphabetizing and comparing text in a particular natural language. These rules define not only a ranking (such as a < b < c) but three levels of priority within the ranking.

For many European languages, the difference between two base letters (*a* and *b*) is a primary difference, the difference between an unaccented and an accented base letter (*õ* and *a*) is secondary, and the difference between an uppercase and lowercase letter (*A* and *a*) is tertiary. These distinctions allow you to set the level of comparison for more sophisticated sorting and searching.

The ICollation interface is based on the protocols in the ANSI C++ standard library collate class, which provides string comparison and hashing functions. The ICollation comparison functions take two strings or substrings and return a value that indicates whether the source string is greater than (later in the alphabet), less than (earlier in the alphabet), or equal to the target string. You can specify the ordering strength of the comparison to control how differences such as case and accents are handled.

You can compare styled text in an IText object, but styling information is ignored.

Collation Subclasses
The collation classes include the abstract base class ICollation, which defines the protocol for language-sensitive string comparison and several concrete subclasses, and ICollationIterator, which lets you iterate through the list of available localized collation objects. The following chart describes some of the collation classes that are available:

Class	Description
IBitwiseCollation	Provides bitwise, language-insensitive string comparison.
ICollation	Provides access to a host-specific collation for a given language as available. Primary class for language-sensitive string comparison.
ICollationIterator	Lets you iterate through the available collation objects.

ICollation provides the protocols you use to create both language-sensitive and language-insensitive collation objects. The ICollation interface is a superset of the interface of the ANSI C++ Standard collate class. Based on the locale you specify, the ICollation::createCollation function can return:

- A host-specific collation object for the specified language or locale
- An IBitwiseCollation object that performs language-insensitive collation
- To request that createCollation return an IBitwiseCollation object, specify the POSIX locale (“POSIX” or “C”).

The following chart describes the interface for ICollation:

Member function	Description
createCollation	A static function that returns the collation object for a specified locale. If you don’t specify a locale, the function returns the collation object for the default locale. createCollation also lets you specify a comparison level. The default is ICollation::kTertiaryDifference.
compare	Returns the result of comparing two strings. The result is returned as an enum value: kSourceEqual, kSourceLess, or kSourceGreater.
strength and setStrength	Strength and setStrength provide access to the collation object’s current ordering strength (primary, secondary, or tertiary).
isEqual, isGreaterThan, and isLessThan	Convenience functions that return a bool value indicating the comparison result of two strings.
transform	Converts an IText into another IText that is compared lexicographically with the original text. Comparing two transformed IText objects returns the same results as comparing the same strings before transformation.

Member function	Description
localeKey	Returns an ILocaleKey indicating the locale the collation object is associated with.
displayName	Returns a displayable name for the object for a specified locale.

Collation Iteration
Use ICollationIterator to iterate through the list of international collation objects currently available on the system.

Ordering Strength
The correct collation for each language or script is determined by a set of rules that define a ranking, from least to greatest, for each character. To allow more comparison options, each character is assigned an ordering priority within the ranking: primary, secondary, or tertiary. For example, in an English collation:

- Base letters represent a primary difference (“a” and “b”)
- Diacritical marks on the same base letter represent a secondary difference (“a” and “Ô”)
- Uppercase and lowercase versions of the same base letter represent a tertiary difference (“a” and “A”)

In English, then, you can implement case-insensitive comparison by setting the ordering strength to kSecondaryDifference. Primary and secondary differences are considered but any tertiary (case) differences are ignored-thus, “pat,” “Pat,” and “PAT” would be considered equivalent strings.

When you create a collation object, you specify an ordering strength that determines whether all differences, both primary and secondary differences, or only primary differences are considered. The types of differences that are considered primary, secondary, and tertiary may vary based on the language you are working with.

This table shows the results for English strings compared with different ordering strengths:

Source	Target	Ordering strength	Comparison result
abc	abc	kPrimaryDifference	kSourceEqual
õbc	abc	kSecondaryDifference	kSourceEqual
Abc	abc	kTertiaryDifference	kSourceEqual
abc	def	kPrimaryDifference	kSourceLess
abc	õbc	kSecondaryDifference	kSourceLess
abc	Abc	kTertiaryDifference	kSourceLess
def	abc	kPrimaryDifference	kSourceGreater
õbc	abc	kSecondaryDifference	kSourceGreater
Abc	abc	kTertiaryDifference	kSourceGreater

When you are using the collation object for the POSIX locale specifying an ordering strength has no effect.

RELATED CONCEPTS

"Chapter 1. International Framework" on page 1
"Locales in Internationalization" on page 2
"Locale Classes" on page 3
Transcoding Classes
"Transcoder Names" on page 20

Instantiate a Collation Object

ICollation is an abstract class that provides the standard interfaces for all collation operations. Subclasses of ICollation provide host-specific collation objects for specific natural languages. IBitwiseCollation is a special portable collation object that provides simple, language-insensitive comparison of Unicode values.

By using the collation classes in conjunction with the locale mechanism, you can implement text sorting and searching features that will work correctly in any user's natural language without needing to know the specific requirements of that language. To create a language-sensitive collation object, call the ICollation::createCollation function. You pass in the locale key corresponding to the language (or more specific locale) of the text you want to collate. For example, you would code the following to get the standard collation for English:

```
ICollation* order =
   ICollation::createCollation(ILocaleKey("EN"));
```

If you use the locale key, createCollation returns the best collation object:

- If the locale key is a valid host locale key, it creates a host-specific collation object.
- If the locale key doesn't represent a valid host or portable locale, the function throws an exception.
- If you don't specify a locale key, the function returns the default collation object for the current locale.

RELATED CONCEPTS

"Collation Classes" on page 8
"Locale Names" on page 5

RELATED TASKS

"Instantiate a Transcoder" on page 21

Perform Case-Insensitive String Comparison

To perform case-insensitive comparison of strings, set the ordering strength of the ICollation object to ignore the level of difference represented by a case difference. Generally cases represent a tertiary difference, although this may differ between languages. This means you set the collation object to consider only primary and secondary differences and ignore tertiary differences.

1. Create the collation object by calling ICollation::createCollation, specifying the desired locale and ordering strength (ICollation::kSecondaryDifference).
2. Call the compare function or one of the helper functions isEqual, isGreaterThan, or isLessThan to compare the two strings.

For example, this code shows how to do case-insensitive comparison of two strings, `text1` and `text2`, using the collation object for U.S. English:

```
// Create a locale key for the U.S. English locale
ILocaleKey usLoc("EN", "US");
// Create the collation object
ICollation*
```

```
   order = ICollation::createCollation
               (usLoc,
                ICollation::kSecondaryDifference);
int result = order->compare(text1, text2);
if (result == ICollation::kSourceEqual)
{
   // strings are equal
   // or only have case differences
}
else
{
   // strings are not equal
}
delete order;
```

RELATED CONCEPTS

"Locale Classes" on page 3
"Locale Names" on page 5
"Collation Classes" on page 8
Ordering Strength

RELATED TASKS

"Perform Language-Sensitive String Comparison"
"Perform Bitwise String Comparison" on page 13
"Use the ICollation::transform Function" on page 14

Perform Language-Sensitive String Comparison

Use ICollation to perform language-sensitive comparison of two Unicode strings. You can use the collation object for the language of the current default locale, or you can specify a particular language or locale.

1. Call ICollation::createCollation to get the collation object for the locale you want. You can indicate both a specific locale and an ordering strength for the collation object. If you don't specify a locale, the functions returns the collation object for the current default locale. If you don't specify an ordering strength, the default is ICollation::kTertiaryDifference.
2. Call ICollation::compare to compare the two strings. The compare function returns an enum value indicating the results of the comparison: kSourceLess (-1), kSourceEqual (0), or kSourceGreater (1). You can also use the functions isEqual, isGreaterThan, or isLessThan. These return a bool.

For example, this code shows how to compare two strings, `text1` and `text2`, using the collation object for the French locale:

```
// Create a locale key for the French locale
ILocaleKey french((IText("FR_FR"));
// Create the collation object,
// using the default ordering strength
ICollation*
   order = ICollation::createCollation(french);
// Compare the strings
ICollation::ETextComparisonResult
result = order->compare(text1, text2);
if (result == ICollation::kSourceEqual)
{
   // strings are equal
}
else if (result == ICollation::kSourceLess)
{
   // text1 is less than text2
}
```

```
else
{
   // text1 is greater than text2
}
delete order;
```

This code shows how to use isEqual to compare the strings, using the collation order for the default locale:

```
ICollation* order = ICollation::createCollation();
if (order->isEqual(text1, text2))
{
   // strings are equal
}
delete order;
```

RELATED CONCEPTS

"Locale Classes" on page 3
"Locale Names" on page 5
"Collation Classes" on page 8
Ordering Strength

RELATED TASKS

"Perform Case-Insensitive String Comparison" on page 11
"Perform Bitwise String Comparison"
"Use the ICollation::transform Function" on page 14

Perform Bitwise String Comparison

IBitwiseCollation performs bitwise (language-insensitive) string comparison, directly comparing the Unicode double-byte character values. Double-byte character sets are used for handling languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by the 256 characters of the single-byte character set.

The following steps show you how to perform a bitwise comparison:

1. Create an IBitwiseCollation object. You can either create the IBitwiseCollation object directly from the IBitwiseCollation constructor, or call ICollation::createCollation, passing in the POSIX ("C") locale.
2. Use compare or one of the helper functions isEqual, isGreaterThan, or isLessThan to compare the strings.

The following code extract performs a bitwise string comparison between the IText objects `text1` and `text2`:

```
// Create a locale key for the POSIX locale
ILocaleKey cLocale("C");
// Create the IBitwiseCollation object
ICollation*
   order = ICollation::createCollation
               (cLocale);
if (order->isEqual(text1, text2))
{
   // strings are equal
}
else
{
   // strings are not equal
}
delete order;
```

Specifying an ordering strength has no effect on the comparison results when using IBitwiseCollation.

RELATED CONCEPTS

"Locale Classes" on page 3
"Locale Names" on page 5
"Collation Classes" on page 8
Ordering Strength
"DBCS and National Language Support" on page 40
"Double-Byte Character Set Support" on page 39

RELATED TASKS

"Perform Case-Insensitive String Comparison" on page 11
"Perform Language-Sensitive String Comparison" on page 12
"Use the ICollation::transform Function"

RELATED REFERENCES

ILocaleKey

Use the ICollation::transform Function

You may want to use the transform function when you are going to compare one string with many other strings, for example, when creating the index for a database.

The result of comparing two transformed strings should be the same as the result of comparing the original strings. For example:

```
ILocaleKey locale("EN", "UK");
ICollation* order = ICollation::createCollation(locale);
xfmSource = order->transform(sourceString);
xfmTarget = order->transform(targetString);
int oldResult = order->compare(sourceString, targetString);
int newResult = xfmSource.compare(xfmTarget);
if (oldResult == newResult)
{
cout << "Transform succeeded.\n";
}
else
{
cout << "Transform did not succeed. \n";
}
delete order
```

RELATED CONCEPTS

"Locales in Internationalization" on page 2
"Locale Classes" on page 3
"Locale Names" on page 5
"Collation Classes" on page 8

RELATED TASKS

"Perform Case-Insensitive String Comparison" on page 11
"Perform Language-Sensitive String Comparison" on page 12
"Perform Bitwise String Comparison" on page 13

RELATED REFERENCES

ICollation
ILocaleKey

Iterate through Available Collation Objects

Use ICollationIterator to iterate through the available collation objects. You may only iterate through host objects; you may not iterate through portable objects. ICollationIterator provides a create function that returns the collation object currently referenced by the iterator.

Follow these steps to use ICollationIterator:

1. Instantiate an ICollationIterator object.
2. Use operator++ to move forward through the list of available collation objects.
3. Use create to get the collation object referenced by the iterator at a given point.

For example, this code shows how to iterate through the available collation objects and instantiate a collation object for French, if it is available:

```
ICollationIterator iter;
bool notFound = true;
ILocaleKey key;
while (notFound && iter)
{
   key.setPOSIXID(iter.localePOSIXID());
   if (key.languageID == "FR")
   {
      ICollation* order = iter.create();
      notFound = false;
   }
   else
   {
      iter++;
   }
}
delete order;
```

RELATED CONCEPTS

"Collation Classes" on page 8

RELATED TASKS

"Iterate through Available Collation Objects"
"Iterate through Available Locales" on page 7
"Iterate through Available Transcoders" on page 26

RELATED REFERENCES

ILocaleKey

Transcoding Classes

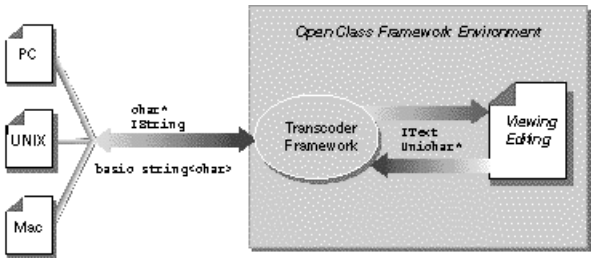
Overview of Transcoding Classes

Transcoding is the process of converting text data between two coded character sets using mapping rules.

The goal of the Open Class libraries is for all text to be encoded in Unicode and manipulated according to the Unicode character encoding standard. Any text you

import from or export to a system that uses a different character encoding scheme must be transcoded so that the text can be manipulated directly on the target system.

The Open Class transcoding classes support conversion of character data to and from Unicode, and a wide variety of other encoding sets and encoding schemes, including ASCII and other ISO standards, This support enables you to import and export text data between Open Class applications and other environments. The Open Class transcoders use the ioc::unichar_t datatype to represent Unicode characters in IText objects and the char datatype to represent non-Unicode characters in IString objects.



The transcoding classes also provide mechanisms for handling characters that do not have obvious mappings between Unicode and another character set. These mechanisms handle both line-breaking characters, which differ between platforms, and *exception characters*. Exception characters are characters that can often be transcoded but do not have a one-to-one mapping. These may include ligature characters, foreign characters, or composed characters.

The following table describes the transcoding classes:

Class	Description
ITranscoder	Primary class defining protocols for transcoding character data between Unicode and any other character encoding standard
ILineBreakConverter	Simple class used to postprocess line-breaking conventions after character data is transcoded into Unicode, or preprocess line-breaking characters before Unicode data is transcoded into char-based data
ICharacterSetIterator	Lets you iterate through the character sets for which transcoders are available

Transcoders
ITranscoder provides the abstract protocol for all transcoders supported by the Open Class system. You create a transcoder by specifying a character set to ITranscoder::createTranscoder, which returns an instance of the ITranscoder subclass supporting that character set.

ITranscoder is currently the only public transcoder class. You access all concrete subclasses through the ITranscoder interfaces.

ITranscoder provides both a simple, high-level interface, which converts between IText and IString instances, and a low-level interface, based on the ANSI C++

Standard codecvt interface, which takes pointers to char and ioc::unichar_t arrays. The high-level functions take two parameters, a char-based IString object and an ioc::unichar_t-based IText object, and convert either to or from Unicode data.

The low-level pointer-based functions let you manipulate char and ioc::unichar_t strings directly. Some of the low-level functions are identical to the interfaces provided by the ANSI C++ standard library codecvt class. They allow you to specify exact ranges of text to transcode and to provide error-recovery mechanisms.

The following table describes the ITranscoder interface:

Member function	Description
createTranscoder	Returns a transcoder for the character encoding set you specify. See the topic Transcoder Names for a list of supported transcoder names. If you don't specify a name, the function returns a transcoder for the current host character set.
toUnicode and fromUnicode	Provide conversion between char and ioc::unichar_t data. Overloads of these functions take IString or char* and IText or ioc::unichar_t*.
result	Returns an enum value that indicates whether the conversion was fully converted, partially converted, or stopped due to an error. This value is also returned by toUnicode and fromUnicode.
unmappedBehavior and setUnmappedBehavior	Let you determine how the transcoder handles exception characters.
setCharSubstitute	Lets you specify a character to be used as a substitute for characters that do not have a mapping from Unicode to the specified character set. The default character is UASCII::kSubstitute (0x1A).
uniCharSubstitute	Returns the character used as a substitute for characters that do not have a mapping from the source character set into Unicode.
characterEncoding	Returns an IText containing the name of the character encoding supported by the transcoder.
characterSet	Returns an IText containing the name of the default encoding for a specified locale.
resetState	Resets the state of the transcoder to ASCII.
Storage query functions	Let you get information about storage requirements so you can manage storage allocation for transcoding operations efficiently.

Line-Breaking Conversion

ILineBreakConverter is a simple class that you use to ensure that line breaking characters are transcoded correctly between Unicode and the target character set. You can use ILineBreakConverter to postprocess strings just converted into Unicode, or preprocess strings before converting them into char data.

The following table describes the ILineBreakConverter interface:

Member function	Description
convertInPlace and convert	Process the line breaks in an IText object according to a specified line-breaking convention.
hostConvention	Returns the line-breaking convention for the current host.

ILineBreakConverter uses the enum ELineBreakConvention to describe different line-breaking conventions. The following describes the current line-breaking conventions:

Constant	Description	Code
kUnicode	Unicode convention	UGeneralPunctuation ::kParagraphSeparator (U+2029)
kCRLF	Windows®, DOS, OS/2® convention	CR LF sequence
kLF	UNIX® convention	LF
kCR	Macintosh System 7 convention	CR
kCRLF_VT	Microsoft Word/Rich Edit convention	CR LF or VT
kHost	Indicates the current host's convention	

ILineBreakConverter uses the following rules to convert between various host line-breaking conventions and Unicode:

Host	Line-breaking convention	Unicode text with host convention	Unicode convention
Win32, OS/2, DOS	CR LF sequence	0x000D 0x000A	U+2029
AIX	LF	0x000A	U+2029
Macintosh	CR	0x000D	U+2029
Word/RichEdit	CR LF sequence	0x000D 0x000A	U+2029
Word/RichEdit	VT	0x000B	U+2028

Character Set Iteration
Use ICharacterSetIterator to iterate through the list of character encoding sets for which transcoders are available on the current system. ICharacterSetIterator returns IText objects that contain the names of supported character sets.

Special Characters
The ITranscoder transcoding functions provide special handling for both line-break and exception characters.

The class ILineBreakConverter provides for conversion between the Unicode paragraph-separator character (U+2029 or UGeneralPunctuation::kParagraphSeparator) and the appropriate line-break

character for a given character set or host. You can use this class to postprocess transcoded strings after conversion into Unicode or to preprocess strings before conversion into char-based formats.

ITranscoder also lets you control how exception characters are handled. Exception characters are characters that do not have a single-character equivalent or that do not exist in the target character set. For example, Greek characters may be used in some environments where they are not part of the native character set, and ligature characters, which by definition combine two characters, are often mapped to a sequence of their individual components. The following table shows some typical cases:

Unicode Name	Unicode sequence	Display	May be mapped to	Control code
LATIN SMALL LIGATURE FI	FB01	[fi]	0066 [f] + 0069 [i]	\xde
GREEK CAPITAL LETTER DELTA	0394	D	² or other	\xc6
GREEK SMALL LETTER PI	03C0	p	* or ¼ or other	\xb9

To specify how you want exception characters to be handled, call the ITranscoder::setUnmappedBehavior function. You can specify a substitution character, in either Unicode or the target character set, or you can specify that the transcoder either skip exception characters or stop the transcoding operation when it reaches one.

By default, transcoders substitute UGeneralFunction::kReplacementCharacter (U+FFFD) for Unicode characters with no mapping, and the ASCII substitution character (UASCII::kSubstitute, or 0x1A) for char characters with no mapping.

Exception Characters
The transcoders let you specify how you want exception characters to be handled. Exception characters are characters for which there are no one-to-one mappings between Unicode and the target character set. Use the EUnmappedBehavior enum to specify one of the following:

kUseSub	Substitutes an equivalent representation in the target character set for characters with no exact mapping
kStop	Stops transcoding when an exception character is detected
kOmit	Skips any exception characters detected

If you don't specify behavior for exception character handling, the transcoder uses EUnmappedBehavior::kUseSub as the default. The substitute characters that are used are:

- UGeneralPunctuation::kReplacementCharacter (U+FFFD) for characters that cannot be transcoded into Unicode
- UASCII::kSubstitute (0x1A) for characters that cannot be transcoded out of Unicode, that is, into the target char-based character set

You can set the char substitute character to another character with the ITranscoder function setCharSubstitute. Whether to display these characters as glyphs or as text strings is left to the host operating system.

Mapping Proximity
When you create a transcoder for a specific character set, you can specify how close the mapping proximity must be between Unicode and the target character set. Use the EMappingProximity enum to specify one of the following:

kExactMapping	Create a transcoder with an exact mapping to the specified character set
kSupersetMapping	Create a transcoder with a character set that is a superset of the specified character set
kCloseMapping	Create a transcoder with a character set as close to the specified character set as possible

If you don't specify a mapping proximity, the transcoder uses EMappingProximity::kSupersetMapping as the default.

RELATED CONCEPTS

Transcoder Names

RELATED TASKS

Instantiate a Transcoder
Iterate through Available Transcoders
Process Line-Breaking Characters
Verify Transcoding Results

Transcoder Names

This table lists the transcoder name for each character encoding set currently supported by the transcoding classes and their corresponding names on various platforms. Use the name listed in the "Open Class Transcoder Name" column to create a transcoder for a particular character encoding set.

Character set	Open Class Transcoder Name	WinNT code page	OS/2 code page	AIX code page
ASCII	US-ASCII	1252	IBM-1252	ISO8859-1
Latin 1/ANSI	ISO-8859-1	1252	IBM-1252	ISO8859-1
EasternEurope	ISO-8859-2	1250 - close	IBM-1250	ISO8859-2
Microsoft Eastern Europe	MSCP-1250	1250	n/a	n/a
Other Latin Set	ISO-8859-3	n/a	n/a	ISO8859-3
Other Latin Set	ISO-8859-4	n/a	n/a	ISO8859-4
ISO Cyrillic	ISO-8859-5	n/a	IBM-1251	ISO8859-5
Microsoft Cyrillic	MSCP-1252	1251	n/a	n/a
ISO Arabic	ISO-8859-6	n/a	IBM-1256	ISO8859-6
Microsoft Arabic	MSCP-1256	1256	n/a	n/a

Character set	Open Class Transcoder Name	WinNT code page	OS/2 code page	AIX code page
ISO Greek	ISO-8859-7	1253 - close	IBM-1253	ISO8859-7
Microsoft Greek	MSCP-1253	1253	n/a	n/a
ISO Hebrew	ISO-8859-8	1255 - close	IBM-1255	ISO8859-8
Microsoft Hebrew	MSCP-1255	1255	n/a	n/a
ISO Turkish	ISO-8859-9	1254 - superset	IBM-1254	ISO8859-9
Microsoft Turkish	MSCP-1254	1254	n/a	n/a
IBM Shift JIS	Shift-JIS	n/a	n/a	IBM-932
Microsoft Shift JIS	Shift-JIS	932 - superset	n/a	n/a
IBM EUC	EUC	n/a	n/a	IBM-eucJP
IBM Simplified Chinese	GB-2312.1980	n/a	n/a	IBM-eucCN
IBM Korean	KSC-5601	n/a	IBM-949	IBM-eucKR
IBM Traditional Chinese	CNS-11643.1986	n/a	IBM-950	IBM-eucTW
8-bit UTF	UTF-8	UTF-8	UTF-8	UTF-8
Macintosh Roman	MSCP-10000	10000	n/a	n/a
MS-DOS US	MSCP-437	437	n/a	n/a
DOS US	IBM-437	n/a	IBM-437	IBM-437
MS-DOS Multilingual	MSCP-850	850	n/a	n/a
DOS Multilingual	IBM-850	n/a	IBM-850	IBM-850

RELATED CONCEPTS

"Chapter 1. International Framework" on page 1
"Locales in Internationalization" on page 2
"Locale Classes" on page 3
"Collation Classes" on page 8
Transcoding Classes

Instantiate a Transcoder

ITranscoder is an abstract class that provides the standard interfaces for all transcoding operations. Transcoders for each character set are implemented in subclasses of ITranscoder. To create a transcoder, call the static function ITranscoder::createTranscoder with a parameter specifying the name of the character set you want to transcode text to or from.

For example, to create a transcoder for the standard Latin1 ISO 8859 character set, you would do the following:

```
ITranscoder* transcoder =
   ITranscoder::createTranscoder("ISO-8859-1");
```

If you don't specify a character set name, the function returns the default transcoder for the current host character set. For example, on a Japanese Windows NT® host, you will get a Shift-JIS transcoder.

RELATED CONCEPTS

Transcoding Classes
"Transcoder Names" on page 20

RELATED TASKS

"Instantiate a Collation Object" on page 11

Convert Text from Character Format to Unicode

Use ITranscoder to convert text data from a char-based format (either char* or IString) to Unicode (either ioc::unichar_t* or IText).

To use ITranscoder to convert from char text data into Unicode text:

1. Call ITranscoder::createTranscoder to create a transcoder for the desired character set. See Transcoder Names for a transcoder name to use. You can also specify a mapping proximity. ITranscoder::kSupersetMapping is the default.
2. Set the behavior for handling exception characters if you want the transcoder to do something other than use substitution characters. You can use ITranscoder::setUnmappedBehavior to specify exception handling behavior.
3. Transcode the text using the toUnicode function.
4. Postprocess the line-breaking characters by calling ILineBreakConverter::convertInPlace or convert.

For example, this code shows how to transcode text from the Microsoft ShiftJIS character set (`charText`) into Unicode:

```
// From character format to Unicode
#include <itrancod.hpp>
#include <iostream.h>
#include <iexcept.hpp>
void main()
{
   try
   {
      // Create the transcoder
      ITranscoder*
         transcoder = ITranscoder::createTranscoder
                        ("Shift-JIS",
                         ITranscoder::kExactMapping);
      // Transcode the string
      IText unicodeText;
      char* charText = "Transcode me!";
      ITranscoder::result
         res = transcoder->toUnicode(charText, unicodeText);
      if (res == codecvt_base::ok)
      {
         cout << "Successful transcoding " << endl
      }
      else
      {
         cout << "Couldn't transcode" << endl;
      }
      // Postprocess any line-breaking characters
      ILineBreakConverter::convertInPlace(unicodeText);
      delete transcoder;
   }
```

```
   catch (IException &ie)
   {
      cout << "Type of exception is: " << ie.name() << endl
           << ie.text() << endl;
   }
}
```

The member function ITranscoder::createTranscoder will throw an exception if it is unable to create a new transcoder object. For example, if you have not installed the Japanese locale and you execute this example, you will see output similar to the following:

```
Type of exception is: IObjectNotFound
Could not create metafile resource.
```

RELATED CONCEPTS

Transcoder Names

RELATED TASKS

Convert Text from Unicode to Character Format
Process Line-Breaking Characters
Convert with ANSI C++ Compatible Transcoding Functions
Verify Transcoding Results

Convert Text from Unicode to Character Format

To convert text data from Unicode to another character encoding standard, use ITranscoder. The Unicode data can be either an ioc::unichar_t* or an IText object. The char data can be either a char* or an IString object.

To convert text from Unicode into another character format:

1. Call ITranscoder::createTranscoder to create a transcoder for the desired character set. See the topic Transcoder Names for the transcoder name to use. You can also specify a mapping proximity. ITranscoder::kSupersetMapping is the default.
2. Set the behavior for handling exception characters if you want the transcoder to do something other than use substitution characters. You can specify the behavior using ITranscoder::setUnmappedBehavior. You can also set specific substitution characters using setCharSubstitute.
3. Preprocess the line-breaking characters by calling ILineBreakConverter::convertInPlace or convert. You must specify the line-breaking convention to use for the non-Unicode text.
4. Transcode the text using the fromUnicode function.

For example, this code shows how to transcode text from Unicode (`unicodeText`) into the ISO-8859-1 (Latin) character set:

```
// From Unicode to character format
#include <itrancod.hpp>
#include <iostream.h>
#include <iexcept.hpp>
void main()
{
   try
   {
      IText unicodeText("Transcode me!");
      // Create the transcoder
      ITranscoder*
         transcoder = ITranscoder::createTranscoder
```

```
                        ("ISO-8859-1");
      // Preprocess any line breaking characters
      ILineBreakConverter::convertInPlace
         (unicodeText, ILineBreakConverter::kHost);
      // Transcode the string
      IString asciiText;
      ITranscoder::result
      res = transcoder->fromUnicode
         (unicodeText,asciiText);
      if (res == codecvt_base::ok)
      {
         cout << "Successful transcoding" << endl
              << unicodeText << endl;
      }
      else
      {
         cout << "Couldn't transcode" << endl;
      }
      delete transcoder;
   }
   catch (IException &ie)
   {
      cout << "Type of exception is: " << ie.name() << endl;
           << ie.text() << endl;
   }
}
```

The member function ITranscoder::createTranscoder will throw an exception if it cannot create a new transcoder object.

RELATED CONCEPTS

Transcoder Names

RELATED TASKS

Convert Text from Character Format to Unicode
Process Line-Breaking Characters
Convert with ANSI C++ Compatible Transcoding Functions
Verify Transcoding Results

Process Line-Breaking Characters

To ensure that line-break characters are correctly transcoded between Unicode and a host character set, use the ILineBreakConverter class, either to preprocess strings before converting them to a host character set or to postprocess strings after converting them into Unicode.

The following steps show how to use ILineBreakConverter:

1. Call the static function convert or convertInPlace. The function convert returns a new IText with the converted string. The function convertInPlace operates directly on the IText that you passed.
2. When preprocessing strings for conversion to a non-Unicode character set, you must specify the convention to use. You can specify a particular convention or ILineBreakConverter::kHost, which uses the convention for the current host.

For example, this code excerpt shows how to postprocess line breaks after conversion into Unicode, then preprocess for conversion back to the original character set:

```
// Transcode into Unicode
transcoder->toUnicode(hostText, unicodeText);
// After transcoding, postprocess the line breaks.
```

```
ILineBreakConverter::convertInPlace(unicodeText);
// Operate on unicodeText
// ...
// Before transcoding out of Unicode,
// preprocess the line breaks
ILineBreakConverter::convertInPlace
   (unicodeText,
    ILineBreakConverter::kHost);
transcoder->fromUnicode(unicodeText, hostText);
```

RELATED CONCEPTS

Transcoding Classes
"Transcoder Names" on page 20

RELATED TASKS

Convert Text from Character Format to Unicode
Convert Text from Unicode to Character Format
Convert with ANSI C++ Compatible Transcoding Functions
"Verify Transcoding Results" on page 26

Convert with ANSI C++ Compatible Transcoding Functions

Instead of transcoding directly between IText and IString objects, you can use pointer-based functions, based on the ANSI C++ Standard codecvt functions, that directly manipulate char* and ioc::unichar_t* data. The following steps describe how to convert from char* data into Unicode:

1. Call ITranscoder::createTranscoder to create a transcoder for the source character set.
2. Establish the range of the source char array to transcode and create a pointer to iterate through the char array during transcoding.
3. Allocate an ioc::unichar_t array to hold the transcoded data and ioc::unichar_t* variables to use as pointers when iterating through the array during transcoding.
4. Transcode using the toUnicode function.

For example, this code shows how to transcode a char string into Unicode:

```
const char* ansiText1 = "An ISO-8859-1 string.";
ITranscoder*
   transcoder = ITranscoder::createTranscoder
                   ("ISO-8859-1");
ioc::unichar_t unicodeText[BUFSIZE];
char* ansiText1_next = NULL;
ioc::unichar_t* unicodeText_end = unicodeText+BUFSIZE;
ioc::unichar_t* unicodeText_next = NULL;
// Transcode the string
ITranscoder::result
res = transcoder->toUnicode
                     (ansiText1,
                      ansiText1+strlen(ansiText1),
                      ansiText1_next,
                      unicodeText,
                      unicodeText_end,
                      unicodeText_next);
if (res == std::codecvt_base::ok)
{
   // Transcoding was successful
}
delete transcoder;
```

RELATED CONCEPTS

Transcoding Classes
Transcoder Names

RELATED TASKS

Convert Text from Character Format to Unicode
Convert Text from Unicode to Character Format
Process Line-Breaking Characters
Verify Transcoding Results

Iterate through Available Transcoders

Use ICharacterSetIterator to iterate through the character sets for which transcoders are available. The iterator returns an IText object containing the name of the character set supported by each transcoder.

The following steps show how to use ICharacterSetIterator:

1. Instantiate an ICharacterSetIterator.
2. Use operator++ to move forward through the list of transcoders.

For example, this code shows how to search through the character sets for which transcoders are available to determine whether a transcoder for for the ISO-8859-7 character set is provided:

```
IText greek("ISO-8859-7");
ICharacterSetIterator iter;
while (iter && *iter != greek)
{
   iter++;
}
if (iter)
{
   // If the iterator was found, construct it
   ITranscoder*
      transcoder = ITranscoder::createTranscoder(iter*);
}
```

RELATED CONCEPTS

Transcoding Classes
"Transcoder Names" on page 20

RELATED TASKS

"Iterate through Available Collation Objects" on page 15
"Iterate through Available Locales" on page 7
"Iterate through Available Transcoders"

RELATED REFERENCES

ITranscoder

Verify Transcoding Results

The ITranscoder transcoding functions return an enum value that indicates how well the transcoder was able to convert the data to the target character set. The following chart describes the possible values:

Value	Description
ok	The conversion was completed.
partial	The source characters were only partially converted.
error	The source contained characters that could not be converted.
noconv	No conversion was needed.

RELATED CONCEPTS

Transcoding Classes

RELATED TASKS

Convert Text from Character Format to Unicode
Convert Text from Unicode to Character Format
"Process Line-Breaking Characters" on page 24
Convert with ANSI C++ Compatible Transcoding Functions

Date and Time Classes

The IDate and ITime classes are independent classes that provide you with data types to store and manipulate date and time information. Because the IDate and ITime classes are independent, when an ITime object's time passes 23:59:59 (24-hour format) or 11:59:59 (12-hour format), it has no effect on the value of any IDate object.

The ITimeStamp class provides you with a data type to store and manipulate timestamp information, where a timestamp represents a specific point in time; for example, combined date and time.

With these classes, you can create date, time, and timestamp objects, and use member functions to do the following:

- Write date, time, or timestamp objects to an output stream
- Access detailed information about dates, times, or timestamps
- Compare dates, times, or timestamps
- Test the characteristics of date or time objects
- Add or subtract days from a date object
- Add or subtract hours, minutes, or seconds from a time or timestamp object
- Convert between date formats or between time formats.

RELATED CONCEPTS

"Dates and Calendars"
Information Functions for IDate Objects
"Time" on page 31
"Information Functions for ITime Objects" on page 31
"Time Stamps" on page 35
"Information Functions for ITimeStamp Objects" on page 35

Dates and Calendars

The IDate class uses Gregorian calendar dates. The Gregorian calendar is in general use and consists of the 12 months, January to December.

IDate also supports the Julian date format, which contains the year in positions 1 and 2, and the day of the year in positions 3 through 5. If the day of the year is less than three digits, zeros are added on the left to increase the size to three digits. For example, February 14, 1965 is 65045 as a Julian date. (February 14 is the 45th day of the year.)

The IDate class returns the names of the days and months in the language defined by the current locale. For information on defining the locale, see the standard C library function setlocale().

RELATED CONCEPTS

"Date and Time Classes" on page 27
Information Functions for IDate Objects
"Time" on page 31
"Information Functions for ITime Objects" on page 31
"Time Stamps" on page 35
"Information Functions for ITimeStamp Objects" on page 35

RELATED TASKS

"Create an IDate Object" on page 29
"Change an IDate Object" on page 29
"Compare and Test IDate Objects" on page 30

Information Functions for IDate Objects

The IDate class defines information functions that you can use to obtain specifics about an IDate object. For example, you can find out what day of the week, month, or year an IDate object's date falls on, or what the name of the day or month is for the current locale. You can also find out what today's date is. The following example shows some of the IDate information functions:

```
//  Information functions for IDate class

#include <iostream.h>
#include <istring.hpp>
#include <idate.hpp>

void main ()
{
   IDate Day1(27,IDate::May,1964);
   cout << Day1.dayName() << " "
        << Day1.monthName() << " "
        << Day1.dayOfMonth() << " out of "
        << IDate::daysInMonth(Day1.monthOfYear(), Day1.year()) << " days in month, "
        << IDate::daysInYear(Day1.year()) << " days in year "
        << Day1.year() <<'.' << endl;
}
```

This program produces the following output:

```
Wednesday May 27 out of 31 days in month, 366 days in year 1964.
```

RELATED CONCEPTS

"Dates and Calendars" on page 27
"Date and Time Classes" on page 27
"Time" on page 31
"Information Functions for ITime Objects" on page 31
"Time Stamps" on page 35
"Information Functions for ITimeStamp Objects" on page 35

Create an IDate Object

You can create an IDate object using different IDate constructors:

```
// Month, day, year
IDate OneDay(IDate::June,30,1994);
// Day, month, year
IDate AnotherDay(23,IDate::April,1961);
// Julian date format
IDate SomeDay(940616);
// Year, day of year
IDate Yesterday(1994,177);
```

The constructors accepting a month use the IDate enumeration Month, whose members are named January through December (the months of the year in English).

RELATED CONCEPTS

"Date and Time Classes" on page 27
"Dates and Calendars" on page 27

RELATED TASKS

"Create an ITime Object" on page 32
"Change an ITime Object" on page 32
"Compare ITime Objects" on page 33
"Write an ITime Object to an Output Stream" on page 34
"Create an ITimeStamp Object" on page 36
"Change an ITimeStamp Object" on page 37
"Compare ITimeStamp Objects" on page 37
"Change an IDate Object"
"Compare and Test IDate Objects" on page 30

Change an IDate Object

You can add days to, or subtract days from, an IDate object. You can also subtract one date from another, in which case the result is the number of days between the two dates. The following example code changes various IDate objects:

```
IDate Day1, Day2;
int NumDays;
Day1=IDate::today();
// Day2 is one day after Day1
Day2=Day1+1;
// Day2 is now three days after Day1
Day2+=2;
// NumDays=3
NumDays=Day2-Day1;
```

You cannot add two IDate objects together because such an addition does not make sense. However, you can add two ITime objects together.

RELATED CONCEPTS

"Date and Time Classes" on page 27
"Dates and Calendars" on page 27

RELATED TASKS

"Create an ITime Object" on page 32
"Change an ITime Object" on page 32

"Compare ITime Objects" on page 33
"Write an ITime Object to an Output Stream" on page 34
"Create an ITimeStamp Object" on page 36
"Change an ITimeStamp Object" on page 37
"Compare ITimeStamp Objects" on page 37
"Create an IDate Object" on page 29
"Compare and Test IDate Objects"

Compare and Test IDate Objects

You can compare two IDate objects to determine whether they are equal, or whether one is later than the other. The following operators are defined: ==, !=, <, <=, >, >=. For example, the following if-statement evaluates true:

```
IDate Day1(IDate::January,1,1994);
IDate Day2(IDate::June,3,1968);
IDate Day3(IDate::July,12,1941);
if ((Day1 > Day 2) && (Day1 != Day3))
{
   // This if-statement evaluates to true
}
```

You can also check whether a particular year is a leap year, or whether a particular combination of day, month, and year is valid. The isLeapYear() function returns true if its integer argument is a leap year.

The isValid() function accepts combinations of day, month, and year (or day of year and year), and returns true if the provided date is valid. For example, the first call to isValid() returns true for the first date below, and the second returns false:

```
if (IDate::isValid(IDate::June, 30, 1990))
{
   // This call to isValid() returns true
}
if (IDate::isValid(1965,366))
{
   // ...
}
else
{
   // This call to isValid returns false.
   // There is no day number 366 in 1965.
}
```

RELATED CONCEPTS

"Date and Time Classes" on page 27
"Dates and Calendars" on page 27

RELATED TASKS

"Create an ITime Object" on page 32
"Change an ITime Object" on page 32
"Compare ITime Objects" on page 33
"Write an ITime Object to an Output Stream" on page 34
"Create an ITimeStamp Object" on page 36
"Change an ITimeStamp Object" on page 37
"Compare ITimeStamp Objects" on page 37
"Create an IDate Object" on page 29
"Change an IDate Object" on page 29

Time

The ITime class refers to time in the 24-hour format by specifying time units (hours, minutes, seconds) past midnight.

To display an ITime object, use either the asString function to convert the ITime object to an IString, or the asText function to convert to an IText object.

To display ITime objects in the 12-hour format, use the asString function with a char* argument of "%r". (This argument is a format string. All format specifiers of the strftime() function of the standard C library are supported by the IString conversion function.)

To display ITime objects in the 12-hour format with an IText object, use the asText function with a Unicode argument of "%r". For example:

```
ITime time;
time.asText(IText("%r"));
```

Note that objects of the ITime class are precise only up to the nearest second, and cannot be used for more precise timings.

RELATED CONCEPTS

"Dates and Calendars" on page 27
"Date and Time Classes" on page 27
Information Functions for IDate Objects
"Information Functions for ITime Objects"
"Time Stamps" on page 35
"Information Functions for ITimeStamp Objects" on page 35

RELATED TASKS

"Create an ITime Object" on page 32
"Change an ITime Object" on page 32
"Compare ITime Objects" on page 33
"Write an ITime Object to an Output Stream" on page 34

Information Functions for ITime Objects

Three of the information functions return an ITime's hour, minute, or second settings; the other information function returns the current time, as determined by the system clock. For example:

```
ITime Time1(ITime::now());
cout << Time1.hours() << " o'clock occurred "
     << Time1.minutes() << " minutes and "
     << Time1.seconds() << " seconds ago." << endl;
```

This displays a result such as the following:

```
12 o'clock occurred 16 minutes and 23 seconds ago.
```

RELATED CONCEPTS

"Dates and Calendars" on page 27
"Date and Time Classes" on page 27
Information Functions for IDate Objects
"Time"
"Time Stamps" on page 35
"Information Functions for ITimeStamp Objects" on page 35

Create an ITime Object

You can create an ITime object and initialize it to a number of seconds past or before midnight, or to a number of hours, minutes, and optionally seconds past midnight. The following examples create ITime objects.

```
// 09:19:16
// 33556 = 9 hours (32400 seconds),
// 19 minutes (1140 seconds),
// 16 seconds (adds up to 33556)
ITime Time1(33556);
// 14:40:44
// (9 hours, 19 minutes and
// 16 seconds BEFORE midnight)
ITime Time2(-33556);
// 12:00:00 (noon)
ITime Time3(12,00);
// 03:03:03
ITime Time4(3,3,3);
```

The constructors translate incorrect times into valid ITime objects using modulo arithmetic. For the seconds past midnight format, any number whose absolute value is greater than or equal to 86400 is divided by 86400, and the remainder is used to calculate the time. For the hours, minutes, and optional seconds format, excess minutes and seconds are added to the hours and minutes values, respectively, and if the hour exceeds 23 it is divided by 24 and the remainder is taken. The following examples show how ITime translates incorrect times into valid ITime objects:

```
// 13:05:56
// (133556 - 86400 = 47156 seconds after midnight)
ITime Time1(133556);
// 10:54:04
// (133556 - 86400 = 47156 seconds BEFORE midnight)
ITime Time2(-133556);
// 12:00:00 (noon)
// (10 hours plus 119 minutes plus 60 seconds)
ITime Time3(10,119,60);
// 09:33:00
// (33 hours - 24 hours = 9 hours)
ITime Time4(33,33);
```

RELATED CONCEPTS

"Date and Time Classes" on page 27
"Time" on page 31

RELATED TASKS

"Change an ITime Object"
"Compare ITime Objects" on page 33
"Write an ITime Object to an Output Stream" on page 34
"Create an ITimeStamp Object" on page 36
"Change an ITimeStamp Object" on page 37
"Compare ITimeStamp Objects" on page 37
"Create an IDate Object" on page 29
"Change an IDate Object" on page 29
"Compare and Test IDate Objects" on page 30

Change an ITime Object

You can add or subtract two times. Four operators are provided: +, +=, -, and -=. The following example shows the use of these operators:

```
ITime Start(12:00), Duration(2:00);
// Done=14:00
ITime Done = Start + Duration;
// Start=12:00 still
Start = Done - Duration;
// Start=14:00
Start += Duration;
// Start=12:00 again
Start -= Duration;
```

RELATED CONCEPTS

"Date and Time Classes" on page 27
"Time" on page 31

RELATED TASKS

"Create an ITime Object" on page 32
"Compare ITime Objects"
"Write an ITime Object to an Output Stream" on page 34
"Create an ITimeStamp Object" on page 36
"Change an ITimeStamp Object" on page 37
"Compare ITimeStamp Objects" on page 37
"Create an IDate Object" on page 29
"Change an IDate Object" on page 29
"Compare and Test IDate Objects" on page 30

Compare ITime Objects

Functions are defined to let you compare ITime objects for equality, inequality, or relative position in time. The following operators are defined: ==, !=, <, <=, >, >=. In the following example, a message is displayed if enough time elapses between the first and second calls to the now() member function:

```
#include <itime.hpp>
#include <iostream>
ITime First(ITime::now());
void main()
{
   ITime Second=ITime::now();
   if (First < Second)
   {
      // Some time has passed
      std::cout << "You must be debugging me!"
                << std::endl;
   }
}
```

This message usually does not print when the program is run outside of a debugging session. However, if you debug the program and step through each line slowly, the message may be displayed, because the first ITime object is initialized during program initialization (before main is called) while the second ITime object is initialized within main.

RELATED CONCEPTS

"Date and Time Classes" on page 27
"Time" on page 31

RELATED TASKS

Write an ITime Object to an Output Stream

ITime defines an output operator that writes an ITime object to an output stream in the format hh:mm:ss. If you want to write the object out in a different format, you should convert the object to an IString using the asString member function. This member function accepts a char* argument containing a format specifier. The format specifier is the same one as used by the C library function strftime. The following program displays some valid specifiers and the output they produce:

```
// Examples of ITime output

#include <istring.hpp>
#include <itime.hpp>
#include <iostream>
// needed for setw()
// to set output stream width
#include <iomanip>
void main() {
   char* FormatStrings[]={
      "%H : %M and %S seconds",
      "%r",
      "%T",
      "%T %Z",
      "%1M past %1I %p"
   };
   // %H, %M, %S - 2 digits for hrs/mins/secs
   // %r         - standard 12-hour clock with am/pm
   // %T         - standard 24 hour clock
   // %Z         - local time zone code
   // %1...      - One digit for hour/minute
   // %p         - am/pm
   // Left-justify output
   sdd::cout.setf(std::ios::left, std::ios::adjustfield);
   // Title text
   std::cout << std::setw(30) << "Format String"
             << setw(40) << "Formatted ITime object"
             << std::endl;
   // Show each time
   for (int i=0;i<5;i++)
   {
      IString
         Formatted=ITime::now().asString(FormatStrings[i]);
      std::cout << std::setw(30) << FormatStrings[i]
                << std::setw(40) << Formatted << std::endl;
   }
}
```

The program produces output that looks like the following:

```
Format String                 Formatted ITime object
%H : %M and %S seconds        16 : 13 and 04 seconds
%r                            04:13:04 PM
%T                            16:13:04
%T %Z                         16:13:04 EST
%1M past %1I %p               13 past 4 PM
```

RELATED CONCEPTS

"Date and Time Classes" on page 27
"Time" on page 31

RELATED TASKS

"Create an ITime Object" on page 32
"Change an ITime Object" on page 32
"Compare ITime Objects" on page 33
"Create an ITimeStamp Object" on page 36
"Change an ITimeStamp Object" on page 37
"Compare ITimeStamp Objects" on page 37
"Create an IDate Object" on page 29
"Change an IDate Object" on page 29
"Compare and Test IDate Objects" on page 30

Time Stamps

An ITimeStamp object can be created from an IDate object, an IDate and ITime object, or a value that represents the number of seconds from the reference date 01/01/2000 00:00:00. If the time stamp is referring to a point in time before the reference date, a negative value must be used.

RELATED CONCEPTS

"Dates and Calendars" on page 27
"Date and Time Classes" on page 27
Information Functions for IDate Objects
"Time" on page 31
"Information Functions for ITime Objects" on page 31
"Information Functions for ITimeStamp Objects"

RELATED TASKS

"Create an ITimeStamp Object" on page 36
"Change an ITimeStamp Object" on page 37
"Compare ITimeStamp Objects" on page 37

Information Functions for ITimeStamp Objects

The ITimeStamp class defines information functions that you can use to obtain specific information about an ITimeStamp object. For example, you can determine the number of seconds separating the ITimeStamp object from the reference date (01/01/2000 00:00:00). You can also find out what the current timestamp is.

Conversion operators have been provided that allow you to convert an existing ITimeStamp object to an IDate object or an ITime object. Once the object has been converted, the IDate or ITime information functions may be then be used.

The following example shows some of the ITimeStamp information functions:

```
// ITimeStamp example
#include <istring.hpp>
#include <itmstamp.hpp>
#include <idate.hpp>
#include <itime.hpp>
int main()
{
   IString::enableInternationalization();
```

```
   ITimeStamp RefDate;
   ITimeStamp TmStamp = ITimeStamp::currentTimeStamp();
   IDate ADate = TmStamp;
   ITime ATime = TmStamp;
   cout << TmStamp << " is " << TmStamp.asSeconds()
        << " seconds apart from" << endl;
   cout << RefDate << endl;
   cout << ATime.hours() << ":" << ATime.minutes() << ":";
   cout << ATime.seconds() << "," << ADate.dayOfYear();
   cout << " days into the year " << ADate.year() << endl;
   return 0;
}
```

This example produces the following output:

```
09/13/1999 17:23:09 is -9.44141e+06 seconds apart from
01/01/2000 00:00:00
17:23:9,256 days into the year 1999
```

The call to IString::enableInternationalization will cause dates to display with 4 digit years.

RELATED CONCEPTS

"Dates and Calendars" on page 27
"Date and Time Classes" on page 27
Information Functions for IDate Objects
"Time" on page 31
"Information Functions for ITime Objects" on page 31
"Time Stamps" on page 35

Create an ITimeStamp Object

You can create an ITimeStamp object using different ITimeStamp constructors:

```
   // Create an IDate object
   IDate ADate(IDate::December, 5, 1963);
   // Create an ITime object
   ITime ATime(10, 11, 12);
   // 12/05/1963 midnight
   ITimeStamp TmStamp1(ADate);
   // 12/05/1963 10:11:12 am
   ITimeStamp TmStamp2(ADate, ATime);
   // 01/01/2000 01:06:40 am
   ITimeStamp TmStamp3(4000.0);
   // 12/31/1999 22:53:20 pm
   ITimeStamp TmStamp4(-4000.0);
   // same as ITimeStamp TmStamp5(0.0);
   // 01/01/2000 00:00:00 am
   ITimeStamp TmStamp5;
```

RELATED CONCEPTS

"Date and Time Classes" on page 27
"Time Stamps" on page 35

RELATED TASKS

"Create an ITime Object" on page 32
"Change an ITime Object" on page 32
"Compare ITime Objects" on page 33
"Write an ITime Object to an Output Stream" on page 34
"Change an ITimeStamp Object" on page 37
"Compare ITimeStamp Objects" on page 37

"Create an IDate Object" on page 29
"Change an IDate Object" on page 29
"Compare and Test IDate Objects" on page 30

Change an ITimeStamp Object

You can add seconds to, or subtract seconds from, an ITimeStamp object. You can also subtract one ITimeStamp object from another, in which case the result is the number of seconds between the two timestamps. The following example code changes various ITimeStamp objects:

```
ITimeStamp TmStamp1, TmStamp2;
double diff;
TmStamp1 = ITimeStamp::currentTimeStamp();
// 4000.0 seconds after TmStamp1
TmStamp2 = TmStamp1 + 4000.0;
// go back 1000.0 seconds
TmStamp2 -= 1000.0;
// should be 3000.0 seconds different
// (if there is no rounding error)
diff = TmStamp2 - TmStamp1;
```

You cannot add two ITimeStamp objects together, as such an addition does not make sense. Also, all the operations are done using floating point arithmetic. As a result, some error due to rounding may occur.

RELATED CONCEPTS

"Date and Time Classes" on page 27
"Time" on page 31

RELATED TASKS

"Create an ITime Object" on page 32
"Change an ITime Object" on page 32
"Compare ITime Objects" on page 33
"Write an ITime Object to an Output Stream" on page 34
"Create an ITimeStamp Object" on page 36
"Compare ITimeStamp Objects"
"Create an IDate Object" on page 29
"Change an IDate Object" on page 29
"Compare and Test IDate Objects" on page 30

Compare ITimeStamp Objects

You can compare two ITimeStamp objects to determine whether they are equal, or whether one is later than the other. The following operators are defined: ==, !=, <, <=, >, and >=.

Since all the operations are done using floating point arithmetic, be aware that some rounding error may occur.

The following example illustrates this point:

```
ITimeStamp TmStamp1(12345.54321);
ITimeStamp TmStamp2 = TmStamp1 + 9753.6802 - 9753.6802;
if (TmStamp1 == TmStamp2)
{
   printf("TmStamp1 == TmStamp2\n");
   printf("TmStamp1 = %30.20f\n", TmStamp1.asSeconds());
   printf("TmStamp2 = %30.20f\n', TmStamp2.asSeconds());
}
```

```
else
{
   printf("TmStamp1 != TmStamp2\n");
   printf("TmStamp1 = %30.20f\n", TmStamp1.asSeconds());
   printf("TmStamp2 = %30.20f\n", TmStamp2.asSeconds());
}
```

This example displays the following output:

```
TmStamp1 != TmStamp2
TmStamp1 =     12345.54321000000000000000
TmStamp2 =     12345.54320999999800000000
```

RELATED CONCEPTS

"Date and Time Classes" on page 27
"Time" on page 31

RELATED TASKS

"Create an ITime Object" on page 32
"Change an ITime Object" on page 32
"Compare ITime Objects" on page 33
"Write an ITime Object to an Output Stream" on page 34
"Create an ITimeStamp Object" on page 36
"Change an ITimeStamp Object" on page 37
"Create an IDate Object" on page 29
"Change an IDate Object" on page 29
"Compare and Test IDate Objects" on page 30

National Language Support and Double-Byte Character Sets

National Language Support

VisualAge C++ provides national language support using the XPG/4 programming model, and using the locale-sensitive functions of the C runtime library.

National Language Support (NLS) is enabled by default. As a result, member functions of the IString, IDate, ITime, and ITimeStamp classes become locale sensitive, in both single-byte character set (SBCS) and double-byte character set (DBCS) environments. The classes provide the following capabilities:

Class	Description
IString	Character string handling in SBCS and DBCS environments.
IDate	Date formatting and manipulation functions.
ITime	Time formatting and manipulation functions.
ITimeStamp	Date and time formatting and manipulation functions.

You can explicitly disable NLS by calling IString::disableInternationalization.

While the interfaces of these classes do not change when you enable NLS, the underlying semantics change to reflect locale requirements. For example, the compare family of IString functions no longer perform bitwise comparisons, but instead perform comparisons based on the string collation sequence defined by the current locale.

RELATED CONCEPTS

"Double-Byte Character Set Support"
"DBCS and National Language Support" on page 40

RELATED TASKS

"Add National Language Support"
"Set the Locale" on page 5

Add National Language Support

To turn on the national language support, use the ICLUI_I18N environment variable:

WIN

OS/2
```
SET ICLUI_I18N=ON
```

AIX
```
export ICLUI_I18N=ON
```

The following statements turn off the national language support:

WIN

OS/2
```
SET ICLUI_I18N=OFF
```

AIX
```
export ICLUI_I18N=OFF
```

The support is on by default.

You can also use the IString class to turn national language support on or off from within your program. The IString class provides three member functions that allow you to programmatically turn internationalization on or off, and test for internationalization:

```
static void enableInternationalization(Boolean enable = true);
static void disableInternationalization();
static bool isInternationalized();
```

RELATED TASKS

"Set the Locale" on page 5

Double-Byte Character Set Support

Objects of the IString class and the I0String class can contain a mixture of single-byte characters and double-byte characters. All member functions allow for the mixture. The searching functions will not match a single-byte character with the second or subsequent byte of a double-byte character. Functions that return substrings will never separate the bytes of a double-byte character.

Although the double-byte characters are supported, you must be careful not to alter the contents of a string in a way that would corrupt the data. For example, the statement:

```
IString[n]='x';
```

would be an error if the nth byte of the IString was part of a double-byte character.

RELATED CONCEPTS

"National Language Support" on page 38
"DBCS and National Language Support"

DBCS and National Language Support

The Data Type and Exception Class Library provides double-byte character set (DBCS) support and national language support (NLS). You can use one source file for your application code and provide DBCS and NLS support by using separate resource files for the languages you support. The benefits of this organization include the following:

- The application is easy to maintain, because a single version of the application is used. This reduces the cost of maintaining your code.
- The application is easy to upgrade because only the source code is upgraded and then linked to the separate language files for different languages. This reduces the time and cost of upgrading your code because different language versions can be generated at the same time.

Because message strings are defined in either resource files or message catalogs, they can be translated easily to your local language without changes to the source code.

You should note the following when creating a DBCS-enabled application:

- String manipulation is DBCS-enabled. The string classes support mixed strings that contain both SBCS and DBCS characters. Use the string testing functions to determine if a character is single byte or double byte.
- The IDBCSBuffer class ensures that the search functions do not match the second or any subsequent bytes of a DBCS character and that the bytes of a DBCS character will not be split.

RELATED CONCEPTS

"National Language Support" on page 38
"Double-Byte Character Set Support" on page 39

RELATED TASKS

"Add National Language Support" on page 39
"Set the Locale" on page 5

Troubleshoot International Objects

If your code throws IInvalidException instances when you try to access international objects, verify that your LOCPATH variable is set correctly. This environment variable identifies the location of international resources on AIX and OS/2 systems, and on Windows systems if the POSIX locales are installed.

If text data you translate into or out of Unicode and then back to the original character set does not look the same as the original data, be aware that some Win32 code pages do not correspond exactly to the Unicode ISO 8859 mapping tables. When transcoding between Unicode and Microsoft code pages, create the transcoder using the following:

- The Microsoft transcoder names provided in "Transcoder Names"
- The mapping proximity ITranscoder::kExactMapping

RELATED CONCEPTS

"Locale Names" on page 5
"Transcoder Names" on page 20
Transcoding Classes

RELATED REFERENCES

ITranscoder

Chapter 2. Text Framework

The classes that are part of the Open Class Text Framework enable the creation, storage, querying, and modification of text strings.

The primary classes provided by the Text Framework are as follows:

Class	Description
IText	A variable-length styled string class you can use for storing styled or unstyled international text. The IText class is optimized for text of any length, from short strings to large text data segments. The length of an IText string is limited by hardware and operating system constraints. For ease of use and consistency, IText stores style data along with character data so that you can use styled characters throughout the Open Class environment. The IText class represents a 2 byte character string. Use this class to create Unicode enabled applications.
ITextBoundary	Implements methods for locating boundaries of characters, words, lines, and sentences.
IString	Can be useful when you do not require the style and international text capabilities provided by IText. The IString represents a single or multibyte string.
Style classes	Can be applied to individual characters, ranges of characters, or paragraphs.
Iterator classes	Provide access to the character data in IText objects.

RELATED CONCEPTS

Text Creation and Manipulation
Text Boundaries
"Text Storage" on page 45
Strings and Buffers
String Formats
Text and Style Run Iteration
"Summary of Text Framework Classes" on page 82

Text Creation and Manipulation

The primary interfaces you need to create and manipulate Unicode text strings, whether styled or unstyled, are provided by IText. It gives you access to the storage mechanism and provides a complete set of protocols for accessing and manipulating both the character and the style data. IText is designed so that you can use it to store unstyled text without the reduction in performance or increase in memory requirements that may be associated with the styling mechanism.

You can create an IText object from another IText, from an IString, or from an array of char or ioc::unichar_t.

IText stores the char-based data internally and transcodes it to Unicode data on demand. At that point, IText assumes that the char-based data is encoded with the character code set of the current locale.

The IText interface parallels the API provided by the ANSI C++ class basic_string. It includes functions for converting between string formats, for manipulating styles, and for working efficiently with the IText storage mechanism.

IText also supports the basic_string interfaces for simple searching and comparison. These functions perform simple bitwise comparisons, and ignore styling information. For language-sensitive comparison of IText strings, use the International Framework's collation classes.

RELATED CONCEPTS

Text Framework
Text Boundaries
Text Storage
Strings and Buffers
String Formats
Text and Style Run Iteration
Summary of Text Framework Classes
Collation Classes
Transcoding Classes
Identifying Text Boundaries

Text Boundaries

Identifying Text Boundaries
The ITextBoundary class implements methods for finding the location of boundaries in text. ITextBoundary is an abstract base class. Instances of ITextBoundary maintain a current position and scan over text returning the index of characters where boundaries occur.

Character Boundaries
Character boundary analysis allows users to interact with characters as they expect to, for example, when moving the cursor through a text string. Character boundary analysis provides correct navigation through strings regardless of how a character is represented. For example, an accented character might be stored as a base character and a diacritical mark, or a single combined character. What users consider to be a character can differ between languages.

Word Boundaries
Word boundary analysis is used by search and replace functions, as well as within text editing applications that allow the user to select words with a double click. Word selection provides correct interpretation of punctuation marks within and following words. Characters that are not part of a word, such as symbols or punctuation marks, have word breaks on both sides.

Line Boundaries
Line boundary analysis determines where a text string can be broken when line-wrapping. The mechanism correctly handles punctuation and hyphenated words.

Sentence Boundaries
Sentence boundary analysis allows selection with correct interpretation of periods within numbers and abbreviations, and trailing punctuation marks such as quotation marks and parentheses.

RELATED CONCEPTS

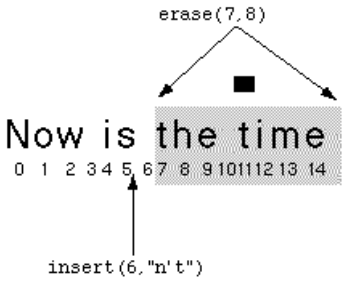
"Chapter 2. Text Framework" on page 43
Text Creation and Manipulation
"Text Storage"
Strings and Buffers
String Formats
Text and Style Run Iteration
"Summary of Text Framework Classes" on page 82

Text Storage

IText is the basic mechanism for storing and manipulating Unicode text strings throughout the Open Class libraries and frameworks. IText encapsulates Unicode characters and any associated styling information, and fully supports mixed style runs. IText keeps the styles with the characters, so you can pass text strings between objects and applications without loss of styling information.

IText is the primary string format supported by the Open Class International Framework. It is suitable for strings of any length, from a few characters to document-length strings. IText was also designed so that you can use it for unstyled strings without incurring the overhead in object size or performance associated with the styling mechanism.

Many IText functions take a range that specifies the subset of characters to operate on. The range is defined by an offset and a character count, where the offset of the first character in the object is 0. To specify an insertion position, specify the offset of the character immediately following the position where the new text will be inserted.



Storage Mechanism
IText manages its own storage. The characters and styles are stored in separate objects that the IText mechanism creates, deletes, and shares transparently. When an IText object is constructed with a char-based format, it stores the char-based data internally and transcodes it to Unicode on demand. IText itself is very small and has very fast copy performance. This allows you to do the following:

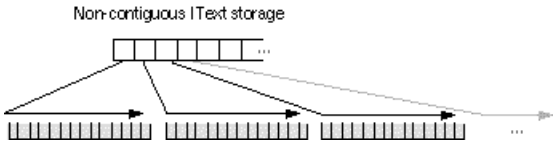
- Include IText objects directly as data members in other objects. Your classes should have pointers or references to IText objects only if you want several objects to all see a change to a particular IText object.

- Pass IText parameters to functions as values rather than references, unless the IText object is owned by another object.
- Return IText objects from functions as return values rather than with fill-in parameters, unless the IText object is owned by another object.
- Use IText objects as local variables—that is, do not use the new and delete operators.

The underlying storage object can be shared by multiple IText instances. It is reference counted and uses copy-on-write semantics. For example, when an IText object is copied, the actual storage is not duplicated—the reference count is simply increased. Note that even while they share storage, two IText objects behave as two distinct objects. They stop sharing storage when one is modified. This is guaranteed to be true even in multithreaded situations.

The framework manages this mechanism for you. However, you should be aware of it when using classes such as IFastTextIterator, which do not consider the underlying storage mechanism.

The storage mechanism handles both short and large strings efficiently. The storage allocation strategy changes dynamically as appropriate for the size of the string. For small strings, the characters are stored in a single, contiguous, heap-allocated array, resized only when necessary. Longer strings are broken up into non-contiguous storage blocks, or chunks, as illustrated in this figure:



The IText function storage_chunk provides access to these chunks of text. You specify a character offset, and the function returns a pointer to the chunk of storage containing that offset. If the string is stored in a single contiguous block, the function returns a pointer to that block.

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43
Text Creation and Manipulation
Text Boundaries
Strings and Buffers
String Formats
Text and Style Run Iteration
"Summary of Text Framework Classes" on page 82

Strings and Buffers

Overview of Strings and Buffers

You can store and manage strings using the string and buffer classes. There are two type of string classes, two types of buffer classes, and two support classes. The two string classes, IString and I0String, are the main classes. The buffer and support classes are used to implement the string classes.

The IString class provides a wide range of string handling capabilities. Many of the IString operators and functions are overloaded to support both IStrings and arrays of characters as return types and arguments. For example, the comparison

operators (==, >, <, >=, <=, !=) all support either two IString operands or one IString and one array of characters operand. The array of characters operand can be on either side of the comparison operator.

If you are using the string classes, DBCS support is nearly automatic and transparent. C-runtime style DBCS support is turned on by default. To disable it on Windows, you have to explicitly call disableInternationalization.

The support classes, IStringEnum and IStringTest, provide data types and testing functions that are used in the string and buffer classes.

String Buffers
When you create an object of a string class, the actual characters that make up the string are not stored in the string object. Instead, the characters are stored in an object of a buffer class.

The use of a buffer object is transparent to you when using the string classes. A correctly sized buffer is automatically created when you create a string object. The buffer is destroyed when a string object is destroyed. When you manipulate or edit a string, you are actually manipulating and editing the buffer object that contains the characters of the string.

String Classes
The string classes define a data type for strings and provide member functions that let you perform a variety of data manipulation and management activities. They provide capabilities far beyond those available with standard C strings and the string.h library functions.

The string classes have the following capabilities:

- String buffers are handled automatically.
- Strings can contain both SBCS and DBCS (including UTF-8) characters.
- Strings can be indexed by character or by word.
- Strings can contain null characters. (There are no restrictions on the contents of a string object.)

Member functions of the string classes allow you to do the following:

- Use strings in input and output
- Access information about strings
- Compare strings
- Test the characteristics of strings
- Search for characters or words within a string
- Manipulate and edit strings
- Convert strings to and from numeric types
- Format strings by adding or removing white space

The two string classes, IString and I0String, are identical except for the method each uses to index its characters. The characters of an IString object are indexed beginning at 1. I0String characters are indexed beginning at 0. The string class you should use depends on which indexing scheme you prefer or find easier to implement.

Objects of IString and objects of I0String can be freely intermixed in a program. Objects of one class can be assigned objects of the other. Arguments that require an

object of one will accept objects of the other. You will only notice a difference between an IString and an I0String when you are using functions that use or return a character index value.

Each IString function has a corresponding I0String function with the same name. The I0String version of each function accepts the same arguments and has the same return type as the IString version except that all parameters of type IString become I0String.

String Comparison
The IStringTest class lets you define the matching function used in the searching and testing functions of the string and buffer classes. When a search string is passed to a searching or testing function, the search string and the string object are compared on a character-by-character basis. The characters are considered to match if they are identical. The IStringTest class allows you to define when characters are considered to match.

For example, you can implement a string test that locates a given occurrence of a particular character in a string:

```
//Using the IStringTest class

#include <istring.hpp>
#include <iostream.h>

class Nth : public IStringTest

{
   char key;               // Specifies the character to look for
   unsigned count;         // Specifies which occurrence to find
   public:
      //
      // Construct an Nth object as follows:
      // 1. Create an IStringTest instance whose function type is user,
      //    with a null character to start;
      // 2. Initialize the count to n
      // 3. Initialize the key to c
      //
      Nth(char c, unsigned n)
      : IStringTest(user,0), count(n), key(c) {}

      //
      // test function: accepts an int (the character to look for)
      // checks if the character matches the key
      // if so, decrements count
      // eventually, count will equal zero if enough matches are found,
      // so "return !count" will return true (-1)
      // otherwise, "return !count" will return a value other than -1

      virtual boolean test (int c) const
         {
         if (c == key)                // if it matches,
         ((Nth*)this)->count--;       // decrement count
         return !count;               // return complement of count
                                      // will be true (-1) if count==0
         }
};

void main()

{
   IString text="this is a test string";
   cout << "The fourth appearance of the letter t in the string:\n"
        << text << '\n' << "is at position "
        << text.indexOf(Nth('t',4)) << endl;
}
```

This program produces the following output:

```
The fourth appearance of the letter t in the string:
this is a test string
is at position 17
```

A derived template class, IStringTestMemberFn, is provided to support the use of the IStringTest class with any function that accepts its objects as an argument.

A constructor for IStringTest accepts a pointer to a C function. The C function must accept an integer as an argument and return a boolean. Such functions can be used anywhere an IStringTest can be used. Note that this is the type of the standard C library functions that check the type of C characters, for example, isalpha() and isupper().

String Indexes
Objects of the string classes are arrays of characters. There are two types of indexes used with the arrays. The first is a character index. Each character is numbered from left to right starting at the number 1 in the IString class and the number 0 in the I0String class. Therefore, in the IString "The dog is brown", the letter "i" has an index value of 9. In the I0String "The dog is brown", the letter "i" has an index value of 8.

The second type of index is the word index. In the word index, each white-space-delimited word is numbered from left to right starting at the number 1. The word index is the same for IString objects and I0String objects. Therefore in the IString "The dog is brown", the word "is" has an index value of 3. In the I0String "The dog is brown", the word "is" also has an index value of 3.

The only difference between objects of the IString class and objects of the I0String class is the starting value for the character index.

To optimize the IString+= operation, the IBuffer/IDBCSBuffer class now allocates memory in 32 bytes chunk(s). To turn this optimization off, you can set the static bool IBuffer::is32BytesAligned to false. By default, if IBuffer::setDefaultBuffer() is called,
the bool is set to false unconditionally.

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43
Text Creation and Manipulation
Text Boundaries
"Text Storage" on page 45
String Formats
Text and Style Run Iteration
"Summary of Text Framework Classes" on page 82
"DBCS and National Language Support" on page 40

String Formats

IText provides full interoperability with the char-based string formats char* and IString, allowing you to do the following:

- Construct IText objects from null-terminated char arrays and from IString objects (IText also provides constructors that take both null-terminated-terminated with UUnicodeSpecial::kNull-and non-terminated ioc::unichar_t arrays)
- Return the characters in an IText as a char array or an IString object

- Use IText where char* or IString is called for
- Use char* or IString where IText is called for

IText uses the transcoding facility provided by the International Framework to convert data between char and Unicode data. IText assumes the char-based format is the default character encoding set for the host. This varies based on the platform and any locale settings for the system.

String Size and Capacity
IText follows the ANSI convention of using size_t (generally equivalent to unsigned long) to represent text lengths and offsets. IText defines two typedefs equivalent to size_t:

Type definition	Description
length_type	Represents a character count. length_type values always represent the number of characters, not the number of bytes. IText also uses a special length_type value, npos, to represent the number of characters between a specified starting offset and the end of the string. For example, a starting offset of 0 and a length of npos refers to the whole string.
offset_type	Represents the position of a character, zero-based from the first character in the string-that is, the position of the first character is at offset_type value 0, and so on.

The following table describes some of the IText functions you can use to access the string's size and storage capacity for the object:

Member function	Description
length and size	Equivalent functions that return a length_type value representing the number of characters currently in the string.
max_size	Returns the maximum number of characters that can be stored in an IText object, currently equivalent to npos - 1.
resize	Lets you reset the size of the string to a specified number of characters, either deleting characters or padding the end of the string as necessary. The default padding character is UUnicodeSpecial::kNull.
capacity	Returns the number of characters for which the IText object has currently allocated storage.
reserve	Can be used to indicate that a specified number of characters are about to be added to the string. The function will preallocate storage if needed and if memory conditions permit it.
empty	Indicates whether the text object contains any character data.

Member function	Description
c_str and data	Return ioc::unichar_t arrays containing the same number of characters as the IText object. c_str always returns a null-terminated array-data may not.

RELATED CONCEPTS

Text Framework
Text Creation and Manipulation
Text Boundaries
Text Storage
Strings and Buffers
Text and Style Run Iteration
Summary of Text Framework Classes

Comparison of IText and IString

IString is the the IBM Open Class primary char-based string class, while IText is the primary ioc::unichar_t (or 2-byte Unicode) based string class. Therefore, one string class is not meant to replace the other. Although IText is usually a better choice, the differences are summarized here:

Use IText:

- if you need to store styled text
- if you want to store international (non-ASCII) text
- if you want powerful language-sensitive comparison
- if you want to store really long runs of text

Note that IText has:

- ties to the IBM Open Class Localization and 2D Graphics Frameworks

Use IString:

- if the object is going to be used with a lot of char*-based system calls, which would force IText to create unnecessary overhead by transcoding repeatedly

Note that IString:

- is used by User Interface Classes

RELATED CONCEPTS

Text Framework
Locales in Internationalization
2D Graphics Overview

RELATED TASKS

Create an IText Object from char or IString Data
Create Strings
Create a Unicode Application

Work with IText Objects

Create an IText Object from char or IString Data

IText provides constructors that initialize IText objects from null-terminated char arrays and from IString objects. The text is transcoded from char data into Unicode data. The framework assumes that the char data is encoded using the default encoding system for the current host.

To create an IText from `char*` or IString data, simply pass the char data to the IText constructor:

```
char* string = "Hello World!";
IText unicodeString(string);
```

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create a Styled Text String"
"Edit Character Data in an IText Object" on page 53
"Extract char* Data from an IText Object" on page 54
"Iterate through Characters in an IText Object" on page 56
"Iterate through Style Runs in an IText Object" on page 57
"Query and Modify Styles in an IText Object" on page 57

Create a Styled Text String

To add styles to characters or paragraphs in an IText object, create the appropriate styles with the correct values and apply them to a specific character range:

1. Instantiate a style object for the style you want to apply, passing in an appropriate value.
2. (Optional) Add the styles to a style set.
3. Call the IText::addStyles function, specifying the range of characters to apply the styles to.

For example:

```
IText styledString("Hello World!");
// Create a style set
ITextStyleSet stylesToAdd;
// Add styles to the style set
stylesToAdd.add(ITextPointSizeStyle(12));
stylesToAdd.add(ITextTypefaceStyle("Courier");
stylesToAdd.add(ITextUnderlineStyle(true));
// Apply styles to the entire string
styledString.addStyles(stylesToAdd, 0, IText::npos);
// Apply another style only to the first word
styledString.addStyles(ITextUnderlineStyle(true), 0, 5);
```

You apply paragraph styles the same way, except you don't have to specify the character range exactly. Specify one or more characters in the paragraph you want to apply styles to, or specify a character range extending across multiple paragraphs. IText automatically extends the paragraph styles to apply to all paragraphs that contain any character offsets you specify. For example:

```
IParagraphJustificationStyle
   pstyle(IParagraphJustificationStyle::kCenter);
styledString.addStyles(pstyles, 0, 1);
```

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create an IText Object from char or IString Data" on page 52
"Edit Character Data in an IText Object"
"Extract char* Data from an IText Object" on page 54
"Iterate through Characters in an IText Object" on page 56
"Iterate through Style Runs in an IText Object" on page 57
"Query and Modify Styles in an IText Object" on page 57

RELATED REFERENCES

IParagraphJustificationStyle
ITextPointSizeStyle
ITextStyleSet
ITextTypefaceStyle
ITextUnderlineStyle

Edit Character Data in an IText Object

IText provides a set of functions for character editing: append and operator+=, insert, insert_and_propagate_styles, replace, and erase. The functions that add characters take either styled or unstyled text. The text that is added maintains its character styling information (raw Unicode characters are considered to be unstyled). Paragraph styles are propagated according to the mechanism described in "Style Propagation" on page 76.

The exception is that characters inserted with the insert_and_propagate_styles function take on the styles of the text they are inserted into.

For example, this code demonstrates some simple text editing functions:

```
IText string("Now is the time for all men to come swiftly to the
aid of the party.");
// Delete "swiftly"
string.erase(36, 8);
// Add "good" before "men"
string.insert(24, IText("good"));
// Change "the party" to "their country"
string.replace(58, 6, IText("ir country"));
// Create a copy of a substring of the text and append more
characters
IText newString = string.substr(0, 15);
newString += IText ("for me to go!");
```

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43
"Style Propagation" on page 76

RELATED TASKS

"Create an IText Object from char or IString Data" on page 52
"Create a Styled Text String" on page 52
"Extract char* Data from an IText Object" on page 54
"Iterate through Characters in an IText Object" on page 56
"Iterate through Style Runs in an IText Object" on page 57
"Query and Modify Styles in an IText Object" on page 57

Extract char* Data from an IText Object

IText provides simple conversion operators that can convert character data in an IText object into a null-terminated char array or an IString object. The text is transcoded from Unicode data into char data. The framework assumes that you want the char data to be encoded in the default encoding system for the current host.

To extract char data from an IText object, simply assign it to a variable of the type you want (char* or IString):

```
IText unicodeString("Hello World!");
const char* charData;
charData = unicodeString;
```

When you extract char data from an IText object, keep in mind that the IText conversion functions return a pointer to an internal storage object. The IText object maintains ownership of this storage, which is why the functions return a const char array. The return value is only guaranteed to be good until the underlying data in the IText is modified. You should not allocate storage to receive the character data, nor should you cast away the const and modify the characters.

If you need a modifiable copy of the character data, allocate your own storage and copy the characters:

```
IText unicodeString("Hello World!");
char* modifiableCharData = new char[unicodeString.length()];
strcpy(modifiableCharData, unicodeString);
// Can modify the char data without affecting the underlying
   IText storage
modifiableCharData[0] = 'J';
const char* moreCharData;
// The underlying IText still contains "Hello World!"
moreCharData = unicodeString;
```

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create an IText Object from char or IString Data" on page 52
"Create a Styled Text String" on page 52
"Edit Character Data in an IText Object" on page 53
"Iterate through Characters in an IText Object" on page 56
"Iterate through Style Runs in an IText Object" on page 57
"Query and Modify Styles in an IText Object" on page 57

Text and Style Run Iteration

Unicode Text Framework provides a set of iterators for accessing the character data in an IText object. These iterators all have the properties of a random-access iterator as described by most recent version of the C++ standard.

Character Iteration

The framework provides five iterators for iterating through the characters in an IText object. These iterators are compatible with the iterator classes provided by the C++ Standard Library:

Class	Description
ITextIterator	Provides iteration through the characters in an IText object.
IReverseTextIterator	Provides backwards iteration through the characters in an IText object.
IConstTextIterator	A variation on ITextIterator that doesn't allow you to modify the characters.
IReverseConstTextIterator	A variation on IReverseTextIterator that doesn't allow you to modify the characters.
IFastTextIterator	Provides faster iteration than the iterators listed above but fewer safety assurances. For example, the iterator is not guaranteed to stay valid if the underlying text is modified.

These iterators all follow the protocol for random-access iterators. You can start the iteration at any position in the IText object and iterate forward, backward, and so on. With the exception of IFastTextIterator, you should not construct iterators directly, but use the IText iterator functions begin, end, rbegin, and rend.

The iterators use the class ICharacterReference to return non-const references to ioc::unichar_t values. This ensures that both the reference-counting mechanism used for IText storage and the paragraph style propagation are not disturbed.

If you want to use IFastTextIterator, construct it from another text iterator. IFastTextIterator is different than the other text iterators in that it doesn't consider the styling mechanism or the shared storage mechanism. You can use IFastTextIterator when you want faster iteration. However, if you want to modify characters while using an IFastTextIterator, you need to ensure that the IText you are iterating over does not share storage with another IText. When using an IFastTextIterator:

- Do not initialize another IText object from the IText under iteration or assign the IText to another IText while the IFastTextIterator is in effect.
- Do not call non-const functions on the IText under iteration while the IFastTextIterator is in effect.
- Keep in mind that IFastTextIterator ignores styles when iterating over styled text. Paragraph styles will not be repropagated, and character styles will not be manipulated with their associated characters.
- Set the willWrite parameter to false if you want to use the iterator for read-only access of characters.

Style Run Iteration

The framework provides an additional iterator, ITextStyleRunIterator, that lets you iterate over the style runs in a styled IText object. A style run is a range of characters with identical styling information—that is, the same set of style objects with the same values. For example, the text string shown here has four style runs:

When using ITextStyleRunIterator, you can specify whether to iterate over character styles, paragraph styles, or both. The iterator considers both types of styles by default.

When you construct the iterator, it points to the first style run.

RELATED CONCEPTS

Text Framework
Text Creation and Manipulation
Text Boundaries
Text Storage
Strings and Buffers
String Formats
Summary of Text Framework Classes

RELATED TASKS

Iterate through Characters in an IText Object
Iterate through Style Runs in an IText Object
Query and Modify Styles in an IText Object

Iterate through Characters in an IText Object

The Unicode Text Framework provides a full set of classes for iterating over characters in an IText object. Use ITextIterator or IConstTextIterator to iterate forwards, or IReverseTextIterator or IReverseConstTextIterator to iterate backwards through the characters in an IText object. You can also use IFastTextIterator when you want faster performance and don't need as many safety checking mechanisms to guarantee the validity of the iterator.

The following steps describe how to use a text iterator:

1. Call IText::begin (or other IText iterator functions as appropriate) to instantiate the iterator.
2. Use the iterator operators ++ and — to iterate forwards and backwards through the characters.

When creating an IConstTextIterator from a non-const IText, do not call ((const IText)txt).begin() or you will get an invalid iterator (due to the temporary IText created by the compiler). Instead,
call ((const IText&)txt).begin() or better yet, use const_cast(text).begin().

For example, this code shows how to use an iterator to strip whitespace characters from the beginning of a text object called `someText`:

```
ITextIterator iter;
// IUnicode::isASpace(*iter) checks whether the
// character currently pointed to by the iterator is
// a space character (a space, tab, and so on).
for (iter = someText.begin();
     iter < someText.end() &&
        !IUnicode::isASpace(*iter);
     ++iter)
{
   someText.erase(someText.begin(), iter);
}
```

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create an IText Object from char or IString Data" on page 52
"Create a Styled Text String" on page 52
"Edit Character Data in an IText Object" on page 53
"Extract char* Data from an IText Object" on page 54
"Iterate through Style Runs in an IText Object"
"Query and Modify Styles in an IText Object"

Iterate through Style Runs in an IText Object

The Unicode Text Framework provides ITextStyleRunIterator for iterating through the style runs in an IText object. To use ITextStyleRunIterator:

1. To create the ITextStyleRunIterator, pass the constructor the IText object you want to iterate over.
2. Use the iterator operators ++ and — to move forward and backward through style runs.
3. Extract the styles on the current style run. The iterator's operator-> lets you call the ITextStyleSet::extract function for the current style run's style set.
4. Use the ITextStyleRunIterator functions runStart and runLength to get the extent of the current style run.

For example, this code shows how to use an iterator to modify the point size for each style run, or add a point-size style if there is none:

```
ITextPointSizeStyle size;
ITextStyleRunIterator iter(someText);
for ( ; iter; ++iter)
{
if (iter->extract(size))
size.setPointSize(size.pointSize() + 3);
else
size.setPointSize(6);
someText.addStyles(size, iter.runStart(), iter.runLength());
}
```

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create an IText Object from char or IString Data" on page 52
"Create a Styled Text String" on page 52
"Edit Character Data in an IText Object" on page 53
"Extract char* Data from an IText Object" on page 54
"Iterate through Characters in an IText Object" on page 56
"Query and Modify Styles in an IText Object"

Query and Modify Styles in an IText Object

The Unicode Text Framework lets you access the styles associated with a given character or range, and add to or remove those styles.

- Use IText::isStyled to determine whether the IText object has any associated styling information.
- Use IText::stylesAt to access the set of styles associated with a specific character range.

- Use IText::maximumStyleSpan to access the range of characters with specific styles.
- Use IText::addStyles or IText::removeStyles to add and remove styles to or from a specific character range.

If you specify a type of style to remove, removeStyles removes any style of that type. It does not look at the value of the style you specified to remove. For example, if you specify to remove ITextBoldfaceStyle(true), removeStyles removes any boldface style whether it is set to true or false.

This code shows how to query the styles at a particular point in the IText object `someText` and modify the style values:

```
offset_type runOffset;
length_type runLength;
const ITextStyleSet* setPtr;
ITextPointSizeStyle size;
// Get the styles on the first style run.
// The offset and length of the run are returned.
setPtr = someText.stylesAt(0, runOffset, runLength)
// If the style run contains a point size style, increment its
   value by 3
if (setPtr.extract(size))
size.setPointSize(size.pointSize() + 3);
// If the style run has no point size style, add one
else
size.setPointSize(6);
// Apply the new point size style to the entire style run
someText.addStyles(size, runOffset, runLength);
```

To look for characters with specific styles, you can use IText::maximumStyleSpan. For example, this code shows how to look for all style runs of bold text, and underline those characters:

```
offset_type cursorOffset = 0;
offset_type spanOffset;
length_type spanLength;
ITextStyleSet set;
ITextBoldfaceStyle bold(true);
ITextUnderlinStyle line(true);
while (cursorOffset < someText.length())
{
// If the current character is boldface, underline the entire
   style run
if (someText.maximumStyleSpan (cursorOffset, bold,
spanOffset, spanLength))
someText.addStyles(line, spanOffset, spanLength);
cursorOffset += spanLength;
}
```

You can also use ITextStyleRunIterator to access style runs.

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43
"Styles and Style Sets" on page 73
Transcoding Classes
Text and Style Run Iteration

RELATED TASKS

"Create an IText Object from char or IString Data" on page 52
"Create a Styled Text String" on page 52

Work with IString Objects

Create Strings

You can create IStrings using constructors. You can use IString constructors that construct null strings, that accept a numeric argument and convert it into a string of numeric characters, or that translate one or more characters into an IString. You can also create a single string out of up to three separate buffers, whose contents are concatenated into the created IString object. The following example shows some of the above ways of creating IString objects:

```
#include <istring.hpp>
void main()
{
   IString Number1(123);    // -> Number1   ="123"
   IString Number2(123.12); // -> Number2   ="123.12"
   IString Character('a');  // -> Character ="a"
   IString String1("a");    // -> String1   ="a"
   IString String2("and");  // -> String2   ="and"
   IString String3("a\0d"); // -> String3   ="a"
}
```

Note that the last string (String3) is initialized with only the first byte of quoted text. The null character in the char* constructor argument is interpreted by the compiler as a terminating null. However, the IString class does support null bytes within strings. To construct String3 as the example intended, you could write:

```
//...
IString String3("and");
String3[2]='\0';
```

If this string is later copied to another string, the null character and following characters are also copied:

```
IString String4=String3;
String4[2]='N';             // -> String4   ="aNd"
```

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Copy Strings" on page 60
"Concatenate Strings" on page 61
"Extend Strings" on page 62
"Format Strings" on page 63
"Determine String Lengths and Word Counts" on page 64
Do String Input and Output
"Find Words or Substrings within Strings" on page 66
"Replace, Insert, and Delete Substrings" on page 67
"Test the Characteristics of Strings" on page 69
"Convert between Strings and Numeric Data" on page 71
"Convert between Strings and Different Base Notations" on page 72

Copy Strings

You can copy IStrings using copy constructors, assignment operators, and substring functions.

The IString assignment operator and copy constructor both copy one string to another string. One of the strings can be an array of characters, or both may be IString objects. The IString assignment operator and copy constructor offer the following advantages over the strcpy and strdup functions provided by the C string.h library:

- When an IString object is copied, a new copy of the string is not made. Instead, the two strings point to the same buffer location. The object is only copied if one of the strings is changed. This means that, for strings that are copied but where neither the source string nor the copy is subsequently changed, performance is improved by the amount of time it would have taken to make the new copy.
- The notation is simple and intuitive. To copy String1 into String2, you simply code String2=String1. With strings defined using the traditional char* method, such an assignment merely copies a pointer to the original string. With IString objects, the assignment copies each byte of the string into the new string.
- You do not have to determine the length of the source string and allocate sufficient storage to store it in the target string before the assignment. IString takes care of allocating the storage for you, whether the target string is being constructed within the assignment or has already been constructed. This reduces the risk of memory violations. In the following example, String2 is constructed and initialized, and then copied to (its original contents are overwritten), while String3 is copy-constructed to contain a copy of String1. Notice that String2's length is extended by the assignment operation.

  ```
  IString String1="A longer string than String2";
  IString String2="A short string";
  IString String3=String1;    // initialized to String1
  String2=String1;            // extended to fit String1
  ```

- The string being copied can contain null characters anywhere within it, and the entire string will be copied.
- If you accidentally create an array of characters without the terminating null, the strcpy function may continue copying past the storage allocated for the string. This can cause storage violations, or, at the least, it can corrupt the data in the target string. The length of IString objects is not determined by a terminating null, so storage violations and corrupt target strings are less likely.

Creating Substrings of Strings

You can use the subString function to return a new IString object containing a portion of another IString. This function lets you create an IString containing the leftmost characters, rightmost characters, or characters in the string's middle. The following example shows calls to subString that create substrings with leftmost, rightmost, or middle characters:

```
// Using the subString method of IString

#include <iostream>
#include <istring.hpp>

void main()
{
   IString All("This is the entire string.");

   // Left -> subString(1, length)
   IString Left=All.subString(1,5);

   // Middle -> (startpos, length)
   IString Middle=All.subString(6,14);
```

```
// Right -> (string length - (substring length - 1) )
IString Right=All.subString(All.length()-6);

std::cout << "<" << All << ">\n"
          << "<" << Left << ">\n"
          << "<" << Middle << ">\n"
          << "<" << Right << ">" << std::endl;
}
```

This program produces the following output:

```
<This is the entire string.>
<This >
<is the entire >
<string.>
```

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create Strings" on page 59
"Concatenate Strings"
"Extend Strings" on page 62
"Format Strings" on page 63
"Determine String Lengths and Word Counts" on page 64
Do String Input and Output
"Find Words or Substrings within Strings" on page 66
"Replace, Insert, and Delete Substrings" on page 67
"Test the Characteristics of Strings" on page 69
"Convert between Strings and Numeric Data" on page 71
"Convert between Strings and Different Base Notations" on page 72

Concatenate Strings

The IString class defines an addition operator (+) to allow you to concatenate two words together. An addition assignment operator (+=) lets you assign the result of the concatenation to the left operand. The copy() member function lets you create an IString consisting of multiple copies of itself or of another string. The following example shows ways of concatenating text onto the start or end of an IString:

```
// Concatenating strings

#include <iostream.h>
#include <istring.hpp>

void main()
{
   IString Str1="Let ";
   IString Str2="us ";
   IString Str3="concatenate ";
   IString Str4="repeatedly ";

   IString Str5=Str1+Str2; // Add Str1 and Str2 and store in Str5
   Str5+=Str3;             // Add Str3 to Str5
   Str4.copy(3);           // Copy Str4 several times onto itself
   Str5+=Str4;             // Add Str4 to Str5
   cout << Str5 << endl;   // Write String 5
}
```

This program produces the following output:

```
Let us concatenate repeatedly repeatedly repeatedly
```

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create Strings" on page 59
"Copy Strings" on page 60
"Extend Strings"
"Format Strings" on page 63
"Determine String Lengths and Word Counts" on page 64
Do String Input and Output
"Find Words or Substrings within Strings" on page 66
"Replace, Insert, and Delete Substrings" on page 67
"Test the Characteristics of Strings" on page 69
"Convert between Strings and Numeric Data" on page 71
"Convert between Strings and Different Base Notations" on page 72

Extend Strings

With arrays of characters, unless you allocate more storage than originally required for a string, you can only extend a string by allocating a new chunk of storage, moving the existing string into the new area, and extending it there.

IString objects are automatically extended for you whenever an IString operator or function requires the extension. This lets you spend more time coding useful function, and less time trying to track down the source of memory violations or data corruption. You can even use the subscript operator to assign a value to a position beyond the end of the string. The following example, by indexing past the end of ShortString, causes the string to be padded with blanks up to position 119, and the letter "a" is added at position 120:

```
IString ShortString="A short string";
ShortString[120]='a';
```

The + and += operators, the assignment operator, and all member functions that change the contents of a string automatically allocate additional storage for the string if that storage is required. This can drastically reduce the amount of string-handling code you need to write.

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create Strings" on page 59
"Copy Strings" on page 60
"Concatenate Strings" on page 61
"Format Strings" on page 63
"Determine String Lengths and Word Counts" on page 64
Do String Input and Output
"Find Words or Substrings within Strings" on page 66
"Replace, Insert, and Delete Substrings" on page 67
"Test the Characteristics of Strings" on page 69
"Convert between Strings and Numeric Data" on page 71
"Convert between Strings and Different Base Notations" on page 72

Format Strings

You can insert padding (white space) into strings so that each string in a group of strings has the same length. The center, leftJustify, and rightJustify functions all do this; their names indicate where they place the existing string relative to the added white space. You provide the final desired length of the string, and the function adds the correct amount of white space (or removes characters if the string is longer than the final length you specify). For example:

```
// Padding IStrings

#include <istring.hpp>
#include <iostream>

void main()
{
   IString s1="Short", s2="Not so short",
           s3="Too long to fit in the desired field length";
   s1.rightJustify(20);
   s2.center(20);
   s3.leftJustify(20);
   std::cout << s1 << '\n' << s2 << '\n' << s3 << std::endl;
}
```

This program produces the following output:

```
               Short
    Not so short
Too long to fit in t
```

If a string is too wide, you can strip leading or trailing blanks using the strip... functions:

```
// Using the strip... functions of IString

#include <istring.hpp>
#include <iostream>

void main()
{
   IString s1, s2, s3, Long=" Lots of space here ";
   s1 = s2 = s3 = Long;
   s1.stripLeading();
   s2.stripTrailing();
   s3.strip();
   std::cout << ">" << Long << "<\n"
             << ">" << s1 << "<\n"
             << ">" << s2 << "<\n"
             << ">" << s3 << "<" << std::endl;
}
```

This program produces the following output:

```
>      Lots of space here     <
>Lots of space here     <
>      Lots of space here<
>Lots of space here<
```

You can also change the case of an IString to all uppercase or all lowercase:

```
// Changing the case of IStrings

#include <iostream>
#include <istring.hpp>

void main()
{
   IString Upper="MANY of THESE are UPPERCASE CHARACTERS";
   IString Lower="Many of these ARE lowercase characters";
```

```
    Upper.change("MANY","NONE").lowerCase();
    Lower.change("Many","None").upperCase();
    std::cout << Upper << '\n' << Lower << std::endl;
}
```

This program produces the following output:

```
none of these are uppercase characters
NONE OF THESE ARE LOWERCASE CHARACTERS
```

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create Strings" on page 59
"Copy Strings" on page 60
"Concatenate Strings" on page 61
"Extend Strings" on page 62
"Determine String Lengths and Word Counts"
Do String Input and Output
"Find Words or Substrings within Strings" on page 66
"Replace, Insert, and Delete Substrings" on page 67
"Test the Characteristics of Strings" on page 69
"Convert between Strings and Numeric Data" on page 71
"Convert between Strings and Different Base Notations" on page 72

Determine String Lengths and Word Counts

You can determine not only the length of a string, but the number of words within the string, or the length of a particular word in the string. The length of a string is not affected by any null characters you insert in the middle of the string. (The strlen function of string.h treats any null character in an array of characters as a terminating null.)

The following descriptions assume that ThisString contains the text "This string has five words".

The length and size functions both return the length of an IString. For example, ThisString.size() returns the value 26, as does ThisString.length().

To determine the number of words in a string, use the numWords member function. For example, ThisString.numWords() returns the value 5.

To determine the length of a particular word, use the lengthOfWord member function. For example, ThisString.lengthOfWord(3) returns the value 3.

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create Strings" on page 59
"Copy Strings" on page 60
"Concatenate Strings" on page 61
"Extend Strings" on page 62
"Format Strings" on page 63

Do String Input and Output

The IString class overloads the input and output operators of the AT&T USL stream class library and the ANSI C++ stream library depending on the user's choice. This allows you to extract IString objects from streams and insert IString objects into them. The input operator reads characters from the input stream until a white-space character or EOF is encountered. The IString class also defines a member function to read a single line from an input stream. The following example shows uses of the input and output operators for IString and the lineFrom function:

```
// Using the IString I/O operators
// and the lineFrom function

#include <istring.hpp>
#include <iostream>

void main()
{
   IString Str1, Str2, Str3;
   Str1="Enter some text:";
   char test[80];

   // Write prompt
   std::cout << Str1;
   // Get input
   std::cin >> Str2;
   // This only reads in one word of text, so we
   // should check to see if this was the only word
   // on the line:
   if (std::cin.peek()!='\n') {
     // there's more text on this line so ignore it
     std::cin.ignore(1000,'\n');
 }
   // Change prompt
   Str1.insert("more ",Str1.indexOf(" text:"));
   // Write prompt again
   std::cout << Str1;
   // Get line of input
   Str3=IString::lineFrom(std::cin,'\n');
   // Write output
   std::cout << "First word of first input: "
             << Str2 << '\n'
             << "Full text of second input: "
             << Str3 << std::endl;
}
```

This example produces the output shown below in regular type, given the input shown in bold:

```
Enter some text:Here is my first string
Enter some more text:Here is my second string
First word of first input: Here
Full text of second input: Here is my second string
```

Note that, although null characters are allowed within an IString object, a null character in an input string is treated as the end of the input, and a null character in an IString being written to an output stream ends the output of that IString.

RELATED CONCEPTS

Text Framework
I/O Stream Classes
The C++ Programming Reference: Streams

RELATED TASKS

Create Strings
Copy Strings
Concatenate Strings
Extend Strings
Format Strings
Determine String Lengths and Word Counts
Find Words or Substrings within Strings
Replace, Insert, and Delete Substrings
Test the Characteristics of Strings
Convert between Strings and Numeric Data
Convert between Strings and Different Base Notations

Find Words or Substrings within Strings

A wide range of functions are available to let you find words, substrings, patterns, or individual characters within a string. You can even do wildcard searches: for example, you can search through a string to find a substring that begins with the letters “Ar” followed by one or more characters, followed by the letters “rk”.

The following example shows a number of the searching functions available for IString objects. Comments describe the type of search operation being carried out.

```
// Searching for substrings
#include <iostream>
#include <istring.hpp>
void main()
{
   IString
      Str1="This string contains some sample text in English.";
   IString
      Str2=Str1.subString(27); // positions 27 and following:
                               // "sample text in English."
   std::cout << "The string under consideration is:\n\n"
             << Str1 << "\n\n";

   // 1. Count the number of occurrences of a substring
   //    within the string
   std::cout << "The substring \"in\" occurs "
             << Str1.occurrencesOf("in")
             << " times in the string.\n";
   // 2. Find the first occurrence of a substring:
   //    (Note that the substring can be a char, char*,
   //    or IString value)
   std::cout << "The letter 'x' first occurs at position "
             << Str1.indexOf('x') << ".\n";
   // 3. Find the first occurrence of any letter of
   //    those specified:
   std::cout
      << "One of the letters q, r, or s first appears at position "
      << Str1.indexOfAnyOf("qrs") << ".\n";
   // 4. Find the first occurrence of any letter
   //    other than those specified:
   std::cout << "The first letter that is not in \"Think\" "
             << "appears at position "
             << Str1.indexOfAnyBut("Think") << ".\n";
   // 5. Find the index of a word
```

```
    std::cout << "The third word starts at position "
              << Str1.indexOfWord(3) << ".\n";
    // 6. Find a match to a phrase, and return the
    // position of the first matching word
    std::cout << "The phrase \"" << Str2
              << "\" starts at word number "
              << Str1.wordIndexOfPhrase(Str2)
              << " of the string.\n";
    // 7. Do a wildcard search to see if the string
    // starts with "Th", contains "co", and ends
    // with "sh."
    std::cout
       << "Does the string match the wildcard search string "
       << "\"Th*co*sh.\"?\n";
    if (Str1.isLike("Th*co*sh.")) std::cout << "Yes.";
    else std::cout << "No.";

    std::cout << std::endl;
}
```

This program produces the following output:

```
The string under consideration is:

This string contains some sample text in English.

The substring "in" occurs 3 times in the string.
The letter 'x' first occurs at position 36.
One of the letters q, r, or s first appears at position 4.
The first letter that is not in "Think" appears at position 4.
The third word starts at position 13.
The phrase "sample text in English." starts at word number 5 of the string.
Does the string match the wildcard search string "Th*co*sh."?
Yes.
```

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create Strings" on page 59
"Copy Strings" on page 60
"Concatenate Strings" on page 61
"Extend Strings" on page 62
"Format Strings" on page 63
"Determine String Lengths and Word Counts" on page 64
Do String Input and Output
"Replace, Insert, and Delete Substrings"
"Convert between Strings and Numeric Data" on page 71
"Convert between Strings and Different Base Notations" on page 72

Replace, Insert, and Delete Substrings

The ability to manipulate the contents of an IString is one of the greatest advantages of the IString class over the traditional method of using string.h functions to manipulate arrays of characters. Consider, for example, a function that perform the following changes on a string. Issues that you need to address when using arrays of characters, but that are handled for you by the IString class, are shown in parentheses:

1. Replace all occurrences of Blue with Yellow (string must be expanded by two characters for each replacement, and text after the replacement must be shifted out).

2. Replace all occurrences of Orange with Pink (string must be shortened by two characters for each replacement).
3. Delete the sixth word of the string. (How are words delimited? By spaces? Carriage returns? Tab characters? What about multiple adjacent whitespace characters?)
4. Insert the word Dark as the fourth word or at the end of the string if the string has fewer than three words. (String must be extended. How are words delimited? Do you add a space before or after the word?).

You can easily handle the above requirements using IString member functions. The sample function fixString() below implements the requirements. Numbered comments correspond to the numbers of the requirements:

```
// Replace, insert, and delete substrings

#include <iostream>
#include <istring.hpp>
void fixString(IString&);
void main()
{
   IString
      Str1="Light Blue and Green are nice colors. ";
   Str1+="But so are Red and Orange.";
   cout << Str1 << endl;
   fixString(Str1);
   cout << Str1 << endl;
}
void fixString(IString &myString) {
   // 1. Change Blue to Yellow
   myString.change("Blue", "Yellow");
   // 2. Change Orange to Pink
   myString.change("Orange", "Pink");
   // 3. Remove words, starting at word 6,
   //    for a total of 1 word.
   myString.removeWords(6,1);
   int Word4=myString.indexOfWord(4);
   // 4. Insert "Dark" as fourth word
   //    or at the end of string if string
   //    has fewer than 4 words. The
   //    insertion occurs 1 byte before
   //    word 4 (otherwise it inserts
   //    in the middle of word 4).
   if (Word4>0)
      myString.insert("Dark ",Word4-1);
   else
      myString+=" Dark";
}
```

This program produces the following output:

```
Light Blue and Green are nice colors. But so are Red and Orange.
Light Yellow and Dark Green are colors. But so are Red and Pink.
```

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create Strings" on page 59
"Copy Strings" on page 60
"Concatenate Strings" on page 61
"Extend Strings" on page 62
"Format Strings" on page 63

Test the Characteristics of Strings

The IString class lets you test your strings to determine characteristics such as the following:

- Whether they represent valid hexadecimal, decimal, or binary values
- Whether they contain only letters, letters and numbers, uppercase letters, lowercase letters, or punctuation characters
- Whether they contain all SBCS or DBCS characters

This list covers only a few of the testing functions provided by IString.

The testing functions return a value of type bool, indicating either true or false for the tested characteristic. For example, the function isBinaryDigits() returns false for the IString value "1101121101". All testing functions return a value of false for null IString.

The testing functions all have names beginning with is..., because they ask a question, such as "is the IString made up only of binary digits?" The following example shows how you can use a subset of these functions:

```
// Evaluating strings using
// the IString is... methods

#include <istring.hpp>
#include <iostream>

void evaluate(IString& StringToTest)
{
      if (StringToTest.isPrintable())
      std::cout << "Evaluating the string "
                << StringToTest << ":" << std::endl;
   else
      std::cout << "Evaluating an unprintable string:"
                << std::endl;
   if (StringToTest.isDigits())
      std::cout << " Contains only digits 0-9."
                << std::endl;
   if (StringToTest.isAlphabetic())
      std::cout << " Contains only alphabetic"
                << " characters." << std::endl;
   if (StringToTest.isAlphanumeric())
      std::cout << " Contains only alphabetic and"
                << " numeric characters." << std::endl;
   if (StringToTest.isBinaryDigits())
      std::cout << " Contains only zeros and ones."
                << std::endl;
   if (StringToTest.isHexDigits())
      std::cout << " Contains only hex digits"
                << " 0-9, a-f, A-F."
                << std::endl;
   if (StringToTest.isControl())
      std::cout << " Contains only ASCII values"
                << " 00-1F, 7F." << std::endl;
   if (StringToTest.isLowerCase())
      std::cout << " Contains only lowercase"
                << " letters a-z." << std::endl;
   if (StringToTest.isUpperCase())
```

```
      std::cout << " Contains only uppercase"
                << " letters a-z." << std::endl;
   if (StringToTest.isSBCS())
      std::cout << " Contains only SBCS characters."
                << std::endl;
}
void main()
{
   IString Str[6];
   Str[0]="12345";            // numeric, hexadecimal
   Str[1]="abcde";            // alphabetic, hexadecimal
   Str[2]="10101";            // numeric, binary
   Str[3]="abCde";            // alphabetic, hexadecimal
   Str[4]="xyz12";            // alphanumeric, lowercase
   Str[5]="\x04\x06\x11\x12"; // control, unprintable
   for (int i=1;i<6;i++) evaluate(Str[i]);
}
```

The output from this program resembles the following. Depending on the code page and character set (ASCII or EBCDIC) of the system you are running the program on, the results may vary.

```
Evaluating the string abcde:
 Contains only alphabetic characters.
 Contains only alphabetic and numeric characters.
 Contains only hex digits 0-9, a-f, A-F.
 Contains only lowercase letters a-z.
 Contains only SBCS characters.
Evaluating the string 10101:
 Contains only digits 0-9.
 Contains only alphabetic and numeric characters.
 Contains only zeros and ones.
 Contains only hex digits 0-9, a-f, A-F.
 Contains only SBCS characters.
Evaluating the string abCde:
 Contains only alphabetic characters.
 Contains only alphabetic and numeric characters.
 Contains only hex digits 0-9, a-f, A-F.
 Contains only SBCS characters.
Evaluating the string xyz12:
 Contains only alphabetic and numeric characters.
 Contains only SBCS characters.
 Evaluating an unprintable string:
 Contains only ASCII values 00-1F, 7F.
 Contains only SBCS characters.
```

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create Strings" on page 59
"Copy Strings" on page 60
"Concatenate Strings" on page 61
"Extend Strings" on page 62
"Format Strings" on page 63
"Determine String Lengths and Word Counts" on page 64
Do String Input and Output
"Find Words or Substrings within Strings" on page 66
"Replace, Insert, and Delete Substrings" on page 67
"Convert between Strings and Numeric Data" on page 71
"Convert between Strings and Different Base Notations" on page 72

Convert between Strings and Numeric Data

The IString class provides a number of as... functions that convert from IString objects to numeric types. You can also convert from numeric types to IString objects by using the versions of the IString constructor that take numeric values as arguments. The following example shows various IString functions that convert between strings and numbers:

```
// Conversion between IString and numeric values
#include <iostream>
#include <istring.hpp>
void main()
{
   // Initialized with a float value
   IString NumStr=1.4512356919E1;
   // Convert to integer value
   int Integer=NumStr.asInt();
   // C++ conversion rules allow asDouble's
   // result to be converted to float
   float Float=NumStr.asDouble();
   // Convert to double value
   double Double=NumStr.asDouble();
   // Assign another integer value
   NumStr=688;
   // Set precision of cout stream
   std::cout.precision(20);
   std::cout << "Integer: " << Integer
             << "\nFloat: " << Float
             << "\nDouble: " << Double
             << "\nString: " << NumStr
             << std::endl;
}
```

This program produces the following output:

```
Integer: 14
Float:   14.512356758117676
Double:  14.512356919
String:  688
```

You can also change the base notation of IString objects containing integer numbers, by using the d2b, d2x or d2c functions, which convert from decimal to binary, hexadecimal, or character representations.

IText and Numberic Formatting

The IText class provides numeric formatting classes through IDecimalFormat, IBinaryFormat, and IHexadecimalFormat.

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create Strings" on page 59
"Copy Strings" on page 60
"Concatenate Strings" on page 61
"Extend Strings" on page 62
"Format Strings" on page 63
"Determine String Lengths and Word Counts" on page 64
Do String Input and Output
"Find Words or Substrings within Strings" on page 66
"Replace, Insert, and Delete Substrings" on page 67

"Test the Characteristics of Strings" on page 69
"Convert between Strings and Different Base Notations"

Convert between Strings and Different Base Notations

You can use the format conversion functions to change the way the data in a string is represented. These functions are overloaded so that each function has two versions. The non-static version replaces the value of the string with the converted value. The static version preserves the original string and returns a new string object containing the converted value. For example:

```
// Changes value of aString
aString.c2b();
// Preserves value of aString
IString binaryDigits = IString::c2b( aString );
```

The conversion functions check the format of the source string to make sure it is compatible with the source format implied by the function name. For example, if you use the b2d function to convert a string from binary to decimal, the function first checks that the string contains only the digits '0' and '1'. If it contains any characters other than those allowed by the source type, the format conversion functions always return 0.

The following example shows the use of the conversion functions. If you examine both the example and the output provided below, you can see how to use the functions.

```
// IString conversion functions
#include <istring.hpp>
#include <iostream>
enum Bases {Bin, Dec, Hex, Char};
IString
   Base[4]={"binary", "decimal", "hex", "character"};
IString NumStr;
void Show(int From, int To, IString& Result)
{
   std::cout << NumStr << " in " << Base[From]
             << " is " << Result << " in " << Base[To]
             << '.' << std::endl;
}
void main()
{
   IString NewStr;
   NumStr="122";
      NewStr=IString::d2b(NumStr); Show(Dec,Bin,NewStr);
      NewStr=IString::d2x(NumStr); Show(Dec,Hex,NewStr);
      NewStr=IString::d2c(NumStr); Show(Dec,Char,NewStr);
   NumStr="Hat";
      NewStr=IString::c2b(NumStr); Show(Char,Bin,NewStr);
      NewStr=IString::c2d(NumStr); Show(Char,Dec,NewStr);
      NewStr=IString::c2x(NumStr); Show(Char,Hex,NewStr);
   NumStr="5F";
      NewStr=IString::x2b(NumStr); Show(Hex,Bin,NewStr);
      NewStr=IString::x2d(NumStr); Show(Hex,Dec,NewStr);
      NewStr=IString::x2c(NumStr); Show(Hex,Char,NewStr);
   NumStr="0110100001101001";
      NewStr=IString::b2d(NumStr); Show(Bin,Dec,NewStr);
      NewStr=IString::b2x(NumStr); Show(Bin,Hex,NewStr);
      NewStr=IString::b2c(NumStr); Show(Bin,Char,NewStr);
}
```

The output from this program resembles the following. Depending on the code page and character set (ASCII or EBCDIC) of the system you are running the program on, the values may vary.

```
122 in decimal is 01111010 in binary.
122 in decimal is 7A in hex.
122 in decimal is z in character.
Hat in character is 010010000110000101110100 in binary.
Hat in character is 4743540 in decimal.
Hat in character is 486174 in hex.
5F in hex is 01011111 in binary.
5F in hex is 95 in decimal.
5F in hex is  in character.
0110100001101001 in binary is 26729 in decimal.
0110100001101001 in binary is 6869 in hex.
0110100001101001 in binary is ÃÐ in character.
```

RELATED CONCEPTS

"Chapter 2. Text Framework" on page 43

RELATED TASKS

"Create Strings" on page 59
"Copy Strings" on page 60
"Concatenate Strings" on page 61
"Extend Strings" on page 62
"Format Strings" on page 63
"Determine String Lengths and Word Counts" on page 64
Do String Input and Output
"Find Words or Substrings within Strings" on page 66
"Replace, Insert, and Delete Substrings" on page 67
"Test the Characteristics of Strings" on page 69
"Convert between Strings and Numeric Data" on page 71

Styles

Styles and Style Sets

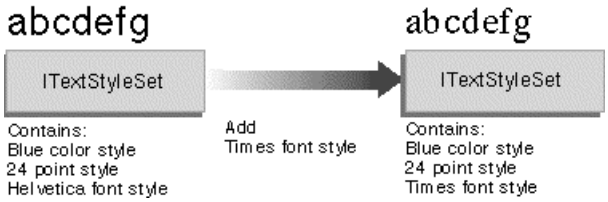
IText uses the classes ITextStyle and ITextStyleSet to encapsulate the styling information associated with a string. ITextStyle is the abstract base class for all text styles provided by the framework. ITextStyleSet provides a mechanism for manipulating styles in groups. Most functions that take styles as parameters allow you to add styles to a range of characters in an IText object either individually (with ITextStyle) or as a group (with ITextStyleSet).

A style object is basically a wrapper for some value that gives the style meaning. The type of value differs for each style. For example, a typeface style encapsulates a string that indicates which typeface to use, such as Times or Helvetica, while the *italic* style encapsulates a boolean that indicates whether the associated characters are *italicized* or not. You can change the value of a style object at any time.

Styles can define information that controls how the text is displayed—for example, the typeface or point size—or some other metadata not related to the display of the text, such as the natural language of the text or whether it is editable. However, styles do not define display mechanisms. It is up to the rendering mechanism to interpret the information in the style objects and display the text according to the associated styling information.

IText uses ITextStyleSet to store the set of styles associated with a particular style run, and you can use it to work with groups of styles. An ITextStyleSet object can contain only one instance of any given style. It cannot, for example, contain two

instances of ITextTypefaceStyle even if one style indicates “Helvetica” and another indicates “Times.” If you add a style to a style set that already contains an instance of that type of style, the new style replaces the old style regardless of their values, as illustrated in this figure:



RELATED CONCEPTS

“Style Classes”
“Style Propagation” on page 76
Character Data

Style Classes

The framework currently supports two types of styles: character styles and paragraph styles. These styles derive from the abstract classes ICharacterStyle and IParagraphStyle, respectively, and are distinguished by how they *propagate*:

- Styles that propagate by character can be applied to any arbitrary range of characters, and maintain their association with those characters regardless of how the surrounding text changes. Character styles include point size and color.
- Styles that propagate by paragraph can be applied only to whole paragraphs—ranges of characters delimited by paragraph separator characters.

Paragraph styles include justification and indentation. For example, if you apply a paragraph style such as a justification style to a range of characters, IText automatically applies it to the entire paragraph (or multiple paragraphs) containing that range. Paragraph styles may be adjusted as text is edited, for example, if paragraph separator characters are added or deleted.

When you create a style object, give it a meaningful value. For example, you might give the value “12” to an ITextPointSizeStyle object or the value IParagraphJustificationStyle**::**kCenter to an IParagraphJustificationStyle object. No default values are provided—if you don’t specify a value when you create a style object, the style’s value remains undefined.

Using an undefined style may be useful, for example, in situations where the value is unimportant (such as calling IText::removeStyles) or where you create the style object to pass it to another function that will fill in the value.

The style classes also provide the ITextStyleSet, which lets you group styles. For example, you could add several styles to a range of characters with a single call to IText. Any IText function that takes an ITextStyle parameter is overloaded to take an ITextStyleSet parameter as well.

Character Style Classes

Character styles derive from ICharacterStyle. ICharacterStyle::propagates returns the value ITextStyle::kPropagateByCharacter. You can apply these character styles to any arbitrary range of characters, from a single character to all characters in an IText object.

This table describes the concrete character styles currently provided by the framework, along with the type of value each contains.

Class	Description	Value
ITextTypefaceStyle	Specifies the typeface to use to display characters.	An IText object encapsulating the name of the typeface, for example, “Helvetica.”
ITextPointSizeStyle	Specifies the point size in which to display characters.	A float representing the point size, for example, “14.”
ITextBoldfaceStyle	Specifies whether characters are displayed in boldface, for example, SampleString.	A bool-true means the characters are displayed in boldface.
ITextItalicStyle	Specifies whether characters are displayed in italics, for example, *SampleString*.	A bool-true means the characters are displayed in italics.
ITextUnderlineStyle	Specifies whether characters are displayed underlined, for example, SampleString.	A bool-true means the characters are underlined.
ITextStrikethroughStyle	Specifies whether characters are displayed with a strikethrough line, for example, SampleString.	A bool-true means the characters are struckthrough.
ITextOutlineStyle	Specifies whether character are displayed in an outline typeface, for example, SampleString.	A bool-true means the characters are outlined.
ITextUneditableStyle	Specifies whether characters can be edited. This is currently the only style that is not related to text display.	A bool-true means the character cannot be edited.
ITextColorStyle	Specifies a color to use to display characters.	Three unsigned char values representing the red, green, and blue components of the color.
ITextSuperSubscriptStyle	Specifies a shift from the baseline, in points, for displaying characters as superscripts or subscripts.	A float representing the baseline shift. A negative value creates a subscript. A positive value create a superscript.

Paragraph Style Classes

Paragraph styles derive from IParagraphStyle. IParagraphStyle::propagates returns the value ITextStyle::kPropagateByParagraph. You can apply paragraph styles only to whole paragraphs, that is, to ranges of characters delimited by UUnicodeSpecial::kParagraphSeparator characters.

If you apply paragraph styles to a range of characters that is less than a paragraph or that intersects multiple paragraphs, IText propagates the paragraph styles to all the relevant paragraphs. The styles also remain associated with the paragraph as the number of characters in the paragraphs changes during editing.

This table describes the concrete paragraph styles currently provided by the framework, along with the type of value each contains.

Class	Description	Value
IParagraphJustificationStyle	Specifies the alignment of the paragraph. Alignment is specified relative to the directionality of the writing system. For example, starting-edge-aligned means on the left for left-to-right languages such as English.	EJustification enum, defining the possible values kStart, kCenter, and kEnd.
IParagraphSpreadStyle	Specifies whether no lines, all lines, or all but the last line of the paragraph are fully justified.	ESpread enum, defining the possible values kNone, kAll, and kNotLastLine.
IParagraphStartIndentStyle	Specifies an indent distance, in points, for the starting edge (the left edge in left-to-right writing systems) of the paragraph.	A float representing the indent distance in points.
IParagraphEndIndentStyle	Specifies an indent distance, in points, for the ending edge (the right edge in left-to-right writing systems) of the paragraph.	A float representing the indent distance in points.
IParagraphFirstIndentStyle	Specifies an indent distance, in points, for the starting edge of the first line of the paragraph.	A float representing the indent distance in points.

Currently the set of styles supported by the Unicode Text Framework is not extensible; you cannot create your own subclasses of ITextStyle, ICharacterStyle, or IParagraphStyle. Future Open Class releases may support extending this style mechanism.

RELATED CONCEPTS

"Styles and Style Sets" on page 73
"Style Propagation"
Character Data

Style Propagation

The Unicode Text Framework uses the propagation of a style to determine how it can be applied to character ranges and how the affected style runs change as the characters are edited. The framework currently defines two types of style propagation:

- *Character-based* propagation
- *Paragraph-based* propagation

Styles that propagate by character can be applied to any arbitrary range of at least one character. The styles remain with these characters regardless of how characters within that range or surrounding that range are edited. Styles applied to a range of characters also propagate to any new text entered into that range.

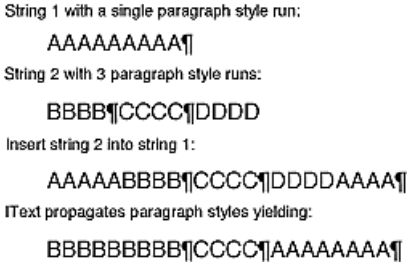
For example, this shows a sequence of possible editing changes:

Styles that propagate by paragraph can be applied only to paragraphs—ranges of characters delimited by paragraph-separator characters (UUnicodeSpecial::kParagraphSeparator). This makes the style propagation mechanism more complicated than it is for character styles, because IText has to repropagate paragraph styles when paragraph separator characters are added or deleted.

The mechanism for propagating styles follows these rules:

- A text string containing no paragraph-separator characters takes on the paragraph styles of the text into which it is inserted.
- A text string bounded on both sides by paragraph-separator characters retains its paragraph styles even when inserted into text containing other paragraph styles.
- A range of text that is inserted into another string and then deleted should have the same paragraph separators and paragraph styles as it did before the editing operations.

IText propagates paragraph styles backwards from the character closest to a paragraph separator back to the beginning of the paragraph. For example, if you have two text strings containing several paragraphs with different paragraph styles:



When propagating paragraph styles, IText recognizes only the UUnicodeSpecial::kParagraphSeparator character (U+2029) to be a paragraph-separator character. When text in other character encoding systems is transcoded into Unicode, you must ensure that it is processed by an ILineBreakConverter object so that the separator characters are transcoded correctly.

RELATED CONCEPTS

"Styles and Style Sets" on page 73
"Style Classes" on page 74
Character Data

RELATED TASKS

"Edit Character Data in an IText Object" on page 53

Character Data

IText provides several functions for accessing specific characters or character ranges.

The framework provides a special class, ICharacterReference, that acts as a simple proxy class to access single ioc::unichar_t values. IText uses ICharacterReference wherever a non-const reference to an ioc::unichar_t is required. ICharacterReference provides a safer character reference mechanism by:

- Filtering out-of-range character references that might cause memory problems
- Protecting the integrity of an IText object's underlying storage mechanism, which might be shared
- Protecting against style propagation problems

ICharacterReference provides an operator that returns a copy of the referenced ioc::unichar_t values.

The following table describes the low-level IText functions for accessing characters:

Member function	Description
operator[] and at	Return a reference to or copy of the character at a specified offset.
at_put	Lets you set the character at a particular offset to a specified character. The ICharacterReference assignment operator also supports this.
substr	Provides an IText object containing a copy of a specified range of characters.

Editing Character Data

IText provides functions for editing character data. You can specify the text to add to the object as:

- Another IText object
- A ioc::unichar_t array (null-terminated or with a specified length)
- A range of an IText defined by a pair of iterators
- A range of an IText defined by an offset and a character count

Unless noted, text added to an IText retains its character styling information (Unicode characters inserted into an IText are considered unstyled). Paragraph styles are propagated as appropriate.

The following table describes the IText interfaces for editing character data:

Member function	Description
operator+= and append	Add a string to the end of the current text.
insert	Inserts a string at a specified offset (you specify the offset of the character after which new text is to be inserted).

Member function	Description
insert_and_propagate_styles	Inserts a string at a specified offset. The inserted string takes on the styles of the surrounding text.
erase	Deletes the characters in a specified range.
replace	Replaces a range of characters with another string. Unlike IString::change, the IText::replace function is not able to perform a full search and replace.

RELATED CONCEPTS

Styles and Style Sets
Style Classes
Style Propagation

Text Display

Unicode Text
The Text and International Frameworks use the Unicode-based class IText as the storage mechanism for underlying text. The text can contain a mixture of runs of different character- and paragraph-styling information.

IGraphicText lets you display a single line of styled text as a static graphic object.

Static Text
The most straightforward mechanism for drawing the styled text represented in an instance of IText is provided by IGraphicText. IGraphicText implements functions inherited from IMGraphic for drawing, performing graphic transformations, and computing the bounds of the graphic in a device-specific manner.

IGraphicText also provides access functions for the text and for the current transform.

After initial construction, IGraphicText lets you use setText to reset the text or the setTransform function to reset the current transform. You can also use the convenience function setLocation to set a new drawing position. This resets the current transform to translate the origin to the specified location. The containing view should call the draw function to redraw the string.

You can use the bounds returned by looseFitBounds to position the graphic. For example, you can create a matrix that rotates the graphic by 45 degrees around the center of the bounding rectangle

For computing the bounding rectangle, the function looseFitBounds returns more accurate results than geometricBounds because geometricBounds returns an approximate bounds based on an untransformed root screen port.

Some graphic transformations may also cause the bounding rectangle to change slightly. You should apply the transformation before computing and using the bounds.

IGraphicText is a subclass of the 2D Graphics class IMGraphic that you use to draw static text strings. IGraphicText draws the styled text in an IText object as a single line of text, beginning at a specified point. IGraphicText also supports basic graphic transformations.

IGraphicText is not intended to be subclassed. To implement your own static text display class, derive directly from classes in the 2D Graphics Framework.

You can specify an IGPoint2D object to represent the anchor position at which to draw the text. For English text, the origin is at the left baseline.

To draw the text, you specify the IGrafPort into which to display the text. For example, this code demonstrates simple usage of IGraphicText:

```
void drawSomeText (const IText& styledText,
   IGPoint2D anchorLocation,
    IGrafPort* displayPort)
{
IGraphicText graphicText(styledText, IGraphicText::kSingleLine);
graphicText.setLocation(anchorLocation);
graphicText.draw(*displayPort);
}
```

RELATED CONCEPTS

Graphics Class Architecture
GrafPorts
IMGraphic Models

Display Text Strings as Graphics

Use IGraphicText to display styled text strings as static graphics. IGraphicText currently displays text as a single line, ignoring text following any new-line characters.

To draw a static text string:

1. Create an IGraphicText instance, specifying the text to draw.
2. Call the IGraphicText::setLocation function, passing in the point representing the position at which to begin drawing. This point is the baseline of the text.
3. Call the IGraphicText::draw function, passing in the port to draw the text to.

For example, this code shows a simple drawContents function for a view that draws static styled text:

```
bool
IHelloWorldView::drawContents (IGrafPort& port) const
{
     IText string("Hello World!");
string.addStyles(ITextTypefaceStyle("Helvetica"));
string.addStyles(ITextPointSizeStyle(18), 0, 5);

IGraphicText hello(string, IGraphicText::kSingleLine);
hello.setLocation(IGPoint2D(30, 30));
hello.draw(port);
return true;
}
```

RELATED CONCEPTS

"Text Display" on page 79

RELATED TASKS

"Apply Graphic Transformations to a Text Graphic"

RELATED REFERENCES

IText
ITextPointSizeStyle
ITextTypefaceStyle

Apply Graphic Transformations to a Text Graphic

IGraphicText supports the IMGraphic functions for graphic transformations, including scaling, rotating, and translating.

To apply a transformation to an existing IGraphicText object:

1. Create an IGrafMatrix representing the transform.
2. Call the IGraphicText::transformBy function to apply the transformation.

   Instead of creating and applying a matrix, you can also use one of the convenience functions rotateBy, translateBy, or scaleBy.

1. Call the IGraphicText::draw function to draw the transformed graphic.

Some transformations, such as rotation, may cause the bounding rectangle to change slightly. Apply the transformation before computing and using the bounds.

For example, this code shows how to center a graphic text object in the IExtendedRootGrafPort port:

```
IManagedPresSpaceHandle presSpace(&drawingArea);
IExtendedRootGrafPort *myDisplayPort
  = new IExtendedRootGrafPort(presSpace);
IText string("Hello World!");
string.addStyles(ITextTypefaceStyle("Helvetica"));
IGraphicText graphic(string, IGraphicText::kSingleLine);
// center the bounding rectangle within the bounding rectangle
// of the port
IGRect2D portBounds = myDisplayPort->worldBounds();
IGRect2D graphicBounds = graphic.looseFitBounds(myDisplayPort);
// apply the transformation
graphic.translateBy(portBounds.center() - graphicBounds.center());
graphic.draw(*myDisplayPort);
delete myDisplayPort;
```

The IGRect2D object `portBounds` represents the rectangular region of the client window.

The IGRect2D object `graphicBounds` represents the smallest rectangle that can fit around the IGraphicText object. The call to looseFitBounds takes into account the transforms and attributes applied to the graphics port, in this case, `myDisplayPort`.

The translateBy function moves a graphic by the specified vector. The function moves the graphic relative to the graphic's origin. Therefore to center a graphic in a graphics port, subtract the center of the graphic from the center of the graphics port. This will take into account the distance between the graphic's origin and center.

The call to draw draws the graphic text onto the graphics port `myDisplayPort`.

RELATED CONCEPTS

"Text Display" on page 79
2D Transformations

RELATED TASKS

"Display Text Strings as Graphics" on page 80

Summary of Text Framework Classes

This table summarizes the Unicode Text Framework classes:

Class	Description
ICharacterReference	Provides a mechanism to reference individual characters without introducing errors in style propagation or reference counting
ICharacterStyle	Abstract base class for all character styles-that is, styles that can be applied to individual characters
IConstTextIterator	A variation on ITextIterator that doesn't allow you to modify the characters
IFastTextIterator	A text iterator that provides faster performance but fewer safety guarantees than ITextIterator
IParagraphEndIndentStyle	Paragraph style specifying the indent distance for the "end" side of the paragraph, for example, the right side of left-to-right text
IParagraphFirstIndentStyle	Paragraph style specifying the indent distance for the start of the first line of the paragraph
IParagraphJustificationStyle	Paragraph style specifying which margin(s) to align the lines of the paragraph to
IParagraphSpreadStyle	Paragraph style specifying which lines of a range of paragraphs are aligned to both margins
IParagraphStartIndentStyle	Paragraph style specifying the indent distance for the "start" side of the paragraph, for example, the left side of left-to-right text
IParagraphStyle	Abstract base class for all paragraph styles-that is, styles that can only be applied to paragraphs and not to individual characters
IReverseConstTextIterator	A variation on IReverseTextIterator that doesn't allow you to modify the characters
IReverseTextIterator	Lets you iterate backwards through the character in an IText object
IText	Encapsulates styled strings of Unicode characters
ITextBoldfaceStyle	Character style specifying whether characters are boldface

Class	Description
ITextColorStyle	Character style specifying the color of characters
ITextItalicStyle	Character style specifying whether characters are italicized
ITextIterator	Lets you iterate through the characters in an IText object
ITextOutlineStyle	Character style specifying whether characters are outlined
ITextPointSizeStyle	Character style specifying the point size of characters
ITextStrikethroughStyle	Character style specifying whether characters have a strikethrough
ITextStyle	Abstract base class for both character and paragraph styles
ITextStyleRunIterator	Lets you iterate through style runs (ranges of text with the same styling information) in an IText object
ITextStyleSet	Class for manipulating a group of styles as a single set
ITextSuperSubscriptStyle	Character style specifying the amount of baseline shift of characters, for creating superscripts and subscripts
ITextTypefaceStyle	Character style specifying the typeface or font of characters
ITextUnderlineStyle	Character style specifying whether characters are underlined
ITextUneditableStyle	Character style specifying whether characters can be modified

RELATED CONCEPTS

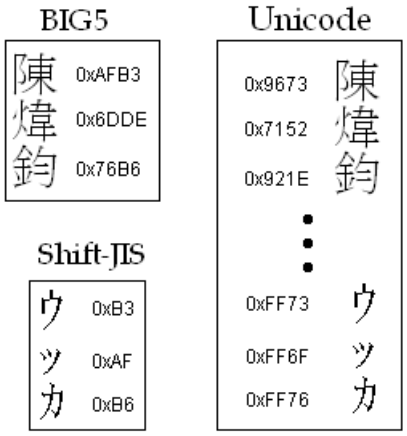
"Chapter 2. Text Framework" on page 43
Text Creation and Manipulation
Text Boundaries
"Text Storage" on page 45
Strings and Buffers
String Formats
Text and Style Run Iteration

Chapter 3. The Unicode Standard

The *Unicode Standard* is a standardized character code designed to encode international texts for display and storage. It uses a unique 16-bit value to represent each individual character. The Unicode standard includes the following:

- Alphabets used in Europe, Africa, and Asia
- Standard characters from China, Japan, Korea, and Taiwan
- Mathematical operators
- Technical symbols

The following diagram illustrates how Unicode assigns a unique 16-bit value to each character:



0xAFB3 represents the 陳 character in BIG5 and the characters ウ ッ in Shift-JIS. Unicode assigns each character with a unique code point. In this case 陳 is assigned with 0x9673 and ウ ッ with 0xFF73 and 0xFF6F.

Although the 16-bit architecture of Unicode can handle more than 65,000 different characters, the Unicode Standard can extend to handle an additional one million characters by the *surrogate extension mechanism*. This mechanism uses two16-bit values to represent one character. The Unicode Standard has not used any of these surrogates. (The current standard contains 38,885 characters.)

The Unicode Standard lets you dynamically compose accented characters. In the Unicode Standard, a character and an accent are separate characters. In other character encodings such as ASCII, you select from a set of accented characters.

The standard supports bidirectional ordering of languages. Bidirectional language ordering occurs when a script uses two or more languages with different dominant directions. For example, a script would have bidirectional language ordering if it mixes Arabic (which reads from right-to-left) with Greek (which reads from left-to-right). The Unicode Standard includes characters that specify a change of direction.

The current implementation of the IBM Open Class Unicode Text framework supports the Unicode Standard without the surrogate extension mechanism.

Unicode Encoding Schemes
The ISO/IEC 10646 (International Organization for Standardization/International Electrotechnical Commission) defines two alternative forms of encoding:

Encoding	Description
UCS-4	The *Universal Character Set coded in 4 bytes* is a 31-bit encoding used to represent each individual character. These coding positions are conceptually divided in 120 *groups* of 256 *planes*, each plane containing 256 *rows* and 256 *columns*.
UCS-2	The *Universal Character Set coded in 2 bytes* is a 16-bit encoding consisting of plane zero, the *Basic Multilingual Plane* (BMP). The Unicode Standard includes all character code values of UCS-2 as well as additional characters.

Transformation Formats
A *transformation format* is used to transform a coding to another coding with a more restrictive numerical range. For example, Unicode consists of characters that have 16 bits, while ASCII characters only have 7 bits. A transformation format would typically transform a Unicode character to one or more bytes so that a system using ASCII characters may understand the Unicode data.

Format	Description
UTF-8	The *UCS Transformation Format, 8-bit form* is a file system safe multi-byte encoding scheme for UCS-2. It is a proper superset of ASCII and it preserves the semantics of a null octet for the C programming language. UTF-8 is mostly used for AIX's file system.
UTF-16	The *UCS Transformation Format for Planes of Group 00*. UTF-16 is the ISO/IEC encoding that is equivalent to the Unicode Standard with the use of surrogates.

Both AIX and NT use UCS-2 as their process codes. Win95/98 is still mostly ASCII based. The AIX platform prefers UTF-8 based functions.

The current implementation of IBM Open Class Unicode Text framework supports both UCS-2 (UTF-16 without the support for surrogates) and UTF-8 encodings for a number of reasons, such as performance, practical interest, and memory burden.

Unicode Character Types
Two data types exist for multilingual characters:

Data type	Description
unsigned short	Data type customarily used for Unicode characters.

Data type	Description
wchar_t	Data type used for extended DBCS (double byte character string).

The IBM Open Class has defined two data types to explicitly represent Unicode as an unsigned short:

Unicode Character Type	Description
UniChar	An unsigned short. The IBM Open Class explicitly declared this typedef so that you can easily distinguish between Unicode characters and unsigned short numbers.
ioc::unichar_t	This data type allow you to switch from unsigned short to wchar_t on systems where wchar_t is two bytes long and supports Unicode. By default this type is defined as an unsigned short. If you define the macro __IOC_USE_WCHAR, ioc::unichar_t becomes a wchar_t.

RELATED CONCEPTS

The Unicode Standard
Unicode Support and the IUnicode Class
Character Values
Character Properties
Summary of Unicode Support Classes

RELATED TASKS

Create a Unicode Application
Identify a Character's Properties
Identify a Character's Script
Find Characters with Specific Properties

Create a Unicode Application

Unicode Enabled Applications
A Unicode enabled IBM Open Class application is one that uses the Unicode interfaces from IBM Open Class. Unicode interfaces are those that either takes an IText object (or reference) or an array of ioc::unichar_t data.

You can mix Unicode and non-Unicode code (in other words, use both IText and IString). However, you may not be able to represent some Unicode (UCS-2) data as ASCII or MBCS data. As a result, you may lose data when converting between encoding schemes.

User interface classes do not support UCS-2 but they do support UTF-8. The 2D graphics classes support UCS-2.

When creating help documents for UTF-8 applications, use HTML help rather than IPF help, since IPF help is not supported under UTF-8 locales. However, the IBM Open Class does not provide classes to work with HTML help.

The IBM Open class provides two string classes: IText and IString. The following chart lists factors that determine which string class you should use:

Factor	Yes	No
Is it a graphical user interface application?	Use either IString or a mixture of IText and IString.	Use IText if you are primarily dealing with UCS-2 data; IString if you are dealing with Multibyte or UTF-8 data.
Do you want ANSI basic_string protocol?	Use IText.	Use either IString or IText.
Are you going to store and manipulate large string objects?	Use IText.	Use either IString or IText.
Do you want to associate styles with your strings (like bold, italic, or color information)?	Use IText.	Use either IString or IText.
Do you need parsing capabilities?	Use IString.	Use either IString or IText.

Unicode Samples
VisualAge C++ provides you with the following samples to help you create Unicode applications:

Name	Location	Description
Unicode Transcoding Sample	samples/intl/transcod	Converts string data between one code page and Unicode
Unicode 2D Graphic Text Sample	samples/intl/graftext	Draws Unicode strings with 2D graphics

RELATED CONCEPTS

The Unicode Standard
Unicode Support and the IUnicode Class
Character Values
Character Properties
Summary of Unicode Support Classes

RELATED TASKS

Identify a Character's Properties
Identify a Character's Script
Find Characters with Specific Properties
Unicode Transcoding Sample
Unicode 2D Graphic Text Sample

Unicode Support and the IUnicode Class

Overview of Unicode Support
Many Open Class classes use the Unicode character encoding standard to represent text data internally. Unicode, a fixed-width, 16-bit character encoding system, contains codes for every character in every major world script, along with a wide

set of symbols, punctuation, and control characters. Because the Unicode system can store and access every character, regardless of its script or natural language, it lets you manipulate text more easily than in environments that require multiple code pages to support different character sets.

The Unicode support classes let you query the properties associated with individual Unicode character values. These properties, provided implicitly by the Unicode character encoding standard, include:

- information about the script (for example, Latin or Cyrillic)
- information about the character's character set (for example, symbols or control characters)
- semantic information, such as whether a character is a digit or is uppercase, lowercase, or uncased.

The IUnicode Class

The primary class in the Unicode support classes is IUnicode, which lets you determine a character's script and character properties. The Unicode support classes also provide a mechanism for referencing specific Unicode character values by name instead of by codepoint values.

IUnicode provides a set of static functions that check a Unicode character, represented by the datatype ioc::unichar_t, for a specific property—for example, querying whether a character is an uppercase character, a digit, or one of the space characters. These functions let you check a character for a specific property without requiring you to know all the possibilities. You can test for a space character, for example, with the IUnicode::IsASpace function without needing to know the full set of Unicode characters used to represent a space.

The class library also provides a set of classes that contain enumerated names for each Unicode character value. These classes correspond to groups of characters based on script or functions: ULatin, UGreek, UDingbats, UMathematicalOperators, and so on. Use the names enumerated in these classes to reference specific Unicode character values.

Character Support

The Unicode character set provides full character coverage for the major scripts listed below, as well as for punctuation, symbols, and control characters. The character set for each script is independent—even if a character appears in multiple scripts, it has a separate code within each script. For example, the character *A* has one code for the Roman alphabet, another code for the Greek alphabet, and yet another code for the Cyrillic alphabet. However, because more than one language may use a given alphabet, the character *A* is represented by the same code for English, French, and, in fact, all languages that use the Roman alphabet.

Supported Scripts

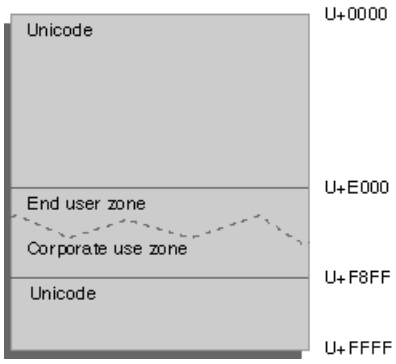
- Arabic
- Armenian
- Bengali
- Cyrillic
- Devanagari
- Georgian
- Greek
- Gujarati

- Gurmukhi
- Han
- Hangul
- Hebrew
- Kana
- Kannada
- Lao
- Malayam
- Oriya
- Roman
- Tamil
- Telugu
- Thai
- Zhuyinfuhao

Reserved Areas
The Unicode standard sets aside a range of characters, from U+E000 to U+F8FF, for the following private uses:

- special characters or sets of characters not included in the Unicode set
- assigning specific semantics to a character

By convention, this area is divided into an end-user zone, which begins at U+E000 and ascends toward higher numbers, and a corporate use zone, which begins at U+F8FF and descends toward lower numbers. The purpose of this convention is to minimize conflicting assignments within the private use area.



RELATED CONCEPTS

The Unicode Standard
Character Values
Character Properties
Summary of Unicode Support Classes

RELATED TASKS

Create a Unicode Application
Identify a Character's Properties
Identify a Character's Script
Find Characters with Specific Properties

Character Values

The class library provides a name, through a set of enumerations, for every character in the Unicode set, with the exception of most of the Han ideographic characters. Names are provided for some particularly significant ideographs, such as digits and the 214 KangXi radicals.

To refer to specific Unicode values, use character names rather than code points. For example, refer to UGeneralPunctuation::kQuestionMark rather than the value U+003F.

Because of the large number of characters, the names are scoped into a set of classes based on script or function. These classes are provided only for referencing the enumerated names they contain; do not use them for any other reason. These classes are listed in the following table. See the enumeration in the referenced header file for specific character names.

Category	**Header file**	**Classes**
General utility characters	iugnrl.hpp	UASCII UCombining UControlCode UGeneralPunctuation ULatin ULatin1 UModifierLetter UUnicodeDigit UUnicodeSpecial
East Asian scripts	iueasia.hpp	UBopomofo UHangulChoseong UHangulJongseong UHangulJungseong UHangulLetter UHangzhouNumeral UHanNumeral UHiragana UIdeographicAnnotation UKangXiRadical UKatakana UKatakanaHiragana
South and Southeast Asian scripts	iusasia.hpp	UBengali UDevanagari UGujarati UGurmukhi UKannada ULao UMalayalam UOriya UTamil UTelugu UThai
Eastern European scripts	iueeuro.hpp	UCoptic UCyrillic UGeorgian UGreek

Category	Header file	Classes
Mideastern scripts	iumeast.hpp	UArabic UArmenian UHebrew
Characters provided for compatibility with other standards	iucmpbty.hpp	UArabicCompatibility UArabicLigature UArmenianSmallLigature UBlocks UBoxDrawings UCircledDigit UCircledHangul UCircledIdeograph UCircledKatakana UCircledLatin UCircledNumber UCJKCompatibility UCJKSquaredAbbreviations UCJKSquaredWords UCNSCompatibility UFullStopDigit UFullStopNumber UFullwidth UHalfwidth UHalfwidthHangulLetter UHangulSyllable UIdeographicTelegraph ULatinSmallLigature UParenthesizedDigit UParenthesizedHangul UParenthesizedIdeograph UParenthesizedLatin UParenthesizedNumber UPresentationFormForVertical URomanNumeral USmallVariants USubscript USuperscript UVulgarFraction
Symbols	iusyms.hpp	UAPLFunctionalSymbol UArrow UCJKSymbols UControlCodePicture UCurrency UDingbats UGeometricShapes UHarpoon ULetterLikeSymbol UMathematicalOperators UMiscellaneousTechnical UOCR UStandardPhonetic UZapfDingbats

Some character names may refer to characters of types other than ioc::unichar_t. In such cases, you may need to cast characters before using interfaces that take ioc::unichar_t parameters.

RELATED CONCEPTS

The Unicode Standard
Unicode Support and the IUnicode Class
Character Properties
Summary of Unicode Support Classes

RELATED TASKS

Create a Unicode Application
Identify a Character's Properties
Identify a Character's Script
Find Characters with Specific Properties

Character Properties

IUnicode provides static member functions that let you access the semantic information provided by the Unicode character standard. These functions let you determine the script of an ioc::unichar_t character or query whether that character has a particular property.

IUnicode includes an enum, EUnicodeScript, that defines the set of scripts supported by Unicode. The function script returns the correct enumerated value that indicates the script of the character. Other functions return a boolean value that indicates whether the character has a particular property.

ICharacterPropertyIterator lets you scan the set of Unicode characters for characters that have a specific set of properties. For example, you might use this class to return a list of punctuation characters for a particular script. The iterator takes a range of character properties, defined by the IUnicode enum ECharacterProperty, and identifies the characters that have the properties in that range.

RELATED CONCEPTS

The Unicode Standard
Unicode Support and the IUnicode Class
Character Values
Summary of Unicode Support Classes

RELATED TASKS

Create a Unicode Application
Identify a Character's Properties
Identify a Character's Script
Find Characters with Specific Properties

Identify a Character's Properties

Use the corresponding IUnicode static member function to determine whether a character has a particular property. For example, this code shows how to iterate through a text object, replacing space characters with hyphens:

```
for (iter = someText.begin();
iter < someText.end() && IUnicode::isASpace(*iter);
++iter);
someText.replace(someText.begin(), iter,
UGeneralPunctuation::kHyphen);
```

RELATED CONCEPTS

The Unicode Standard
Character Properties

RELATED TASKS

Create a Unicode Application
Identify a Character's Script
Find Characters with Specific Properties

Identify a Character's Script

Use the IUnicode::script function to determine the script of a character. Scripts are identified by the IUnicode::EScript enumeration. For example, this code shows how to determine whether a character at a specified offset in a text object is from the Roman script:

```
ioc::unichar_t c = someText.at(0);
if IUnicode::script(c) = IUnicode::kRoman
{
// Character is of the Roman script...
}
```

RELATED CONCEPTS

The Unicode Standard
Unicode Support and the IUnicode Class

RELATED TASKS

Create a Unicode Application
Identify a Character's Properties
Find Characters with Specific Properties

Find Characters with Specific Properties

Use the class ICharacterPropertyIterator to find the set of Unicode characters with particular properties. This iterator identifies the set of characters with a specified range of properties, based on the IUnicode::ECharacterProperty enum.

1. Create an ICharacterPropertyIterator, specifying the range of properties to identify.
2. Use operator++ to advance the iterator.
3. Use operator* to access the character currently referenced by the iterator.
4. Use operator bool to determine when the iterator is at the end of the list.

For example, this code shows how to iterate through the Unicode character set and build a list of cased letters (characters with either a lowercase or an uppercase property):

```
ioc::unichar_t longList[65536];
int n = 0;
// Create the iterator
ICharacterPropertyIterator iter(IUnicode::kUpperCaseLetter,
IUnicode::kLowerCaseLetter);
while(iter)
{
// Access the next character with case properties
longList[n] = iter*;
```

```
// Advance the iterator
iter++;
n++;
}
```

RELATED CONCEPTS

The Unicode Standard
Character Properties

RELATED TASKS

Create a Unicode Application
Identify a Character's Properties
Identify a Character's Script

Summary of Unicode Support Classes

This table lists the Unicode support classes:

Class	Description
ICharacterPropertyIterator	Lets you identify the set of Unicode characters with specific character properties
IUnicode	Provides access to the script and character properties associated with a Unicode character
UAPLFunctionalSymbol	Enumerates names for APL functional symbols in Unicode
UArabic	Enumerates names for characters in the Arabic script, excluding ligatures and compatibility variants
UArabicCompatibility	Enumerates names for characters required for compatibility with older Arabic character sets
UArabicLigature	Enumerates names for Arabic ligatures
UArmenian	Enumerates names for characters in the Armenian script, excluding ligatures
UArmenianSmallLigature	Enumerates names for ligatures of lowercase Armenian letters
UArrow	Enumerates names for arrow symbols in Unicode
UASCII	Enumerates names for Unicode characters from the ASCII character set
UBengali	Enumerates names for characters in the Bengali script
UBlocks	Enumerates names for Unicode characters used in some terminal applications to draw blocks and filled-in shapes on the screen
UBopomofo	Enumerates names for characters in the Bopomofo (or Zhuyinfuhao) alphabet used to write Chinese phonetically
UBoxDrawings	Enumerates names for characters used to draw boxes on the screen

Class	Description
UCircledDigit	Enumerates names for digits inside circles
UCircledHangul	Enumerates names for Hangul (Korean) characters inside circles
UCircledIdeograph	Enumerates names for East Asian ideographs inside circles
UCircledKatakana	Enumerates names for Katakana characters inside circles
UCircledLatin	Enumerates names for Latin characters inside circles
UCircledNumber	Enumerates names for numbers inside circles
UCJKCompatibility	Enumerates names for East Asian ideographs required for compatibility with existing national and industrial standards
UCJKSquaredAbbreviations	Enumerates names for Latin abbreviations written in square blocks in Japanese
UCJKSquaredWords	Enumerates names for Japanese words or phrases written in square blocks
UCJKSymbols	Enumerates names for symbols used in the writing of East Asian languages or from East Asian character sets
UCNSCompatibility	Enumerates names for characters required for compatibility with CNS 11643-1986
UCombining	Enumerates names for certain combining characters used in Unicode
UControlCode	Enumerates names for control codes defined in Unicode
UControlCodePicture	Enumerates names for graphic symbols used to represent standard control codes when discussing them (as opposed to executing them)
UCoptic	Enumerates names for characters in the Coptic script
UCurrency	Enumerates names for currency symbols
UCyrillic	Enumerates names for characters in the Cyrillic script
UDevanagari	Enumerates names for characters in the Devanagari script
UDingbats	Enumerates names for miscellaneous dingbat symbols encoded in Unicode
UFullStopDigit	Enumerates names for digits followed by periods
UFullStopNumber	Enumerates names for numbers followed by periods
UFullwidth	Enumerates names for extra-wide copies of certain Unicode characters
UGeneralPunctuation	Enumerates names for punctuation characters within Unicode, such as spaces, dashes, and so on

Class	Description
UGeometricShapes	Enumerates names for Unicode characters providing prototypes for various geometric shapes
UGeorgian	Enumerates names for characters in the Georgian script
UGreek	Enumerates names for characters in the Greek script
UGujarati	Enumerates names for characters in the Gujarati script
UGurmukhi	Enumerates names for characters in the Gurmukhi script
UHalfwidth	Enumerates names for extra-narrow copies of certain Unicode characters
UHalfwidthHangulLetter	Enumerates names for half-width Hangul letters
UHangulChoseong	Enumerates names for initial conjoining Korean letters (jamos)
UHangulJongseong	Enumerates names for final consonant conjoining Korean letters (jamos)
UHangulJungseong	Enumerates names for medial vowel conjoining Korean letters (jamos)
UHangulLetter	Enumerates names for non-conjoining forms of Korean letters (jamos)
UHangulSyllable	Enumerates names for precomposed Hangul syllables
UHangzhouNumeral	Enumerates names for Hangzhou-style numerals used in East Asia
UHanNumeral	Enumerates names for East Asian ideographs used to write numerals
UHarpoon	Enumerates names for harpoon-like symbols within Unicode
UHebrew	Enumerates names for characters in the Hebrew script
UHiragana	Enumerates names for the Hiragana syllabary used in writing Japanese
UIdeographicAnnotation	Enumerates names for Kanbun marks used in Japanese to indicate the Japanese reading order of classical Chinese texts
UIdeographicTelegraph	Enumerates names for East Asian symbols from telegraph codes
UKangXiRadical	Enumerates names for the traditional 214 radicals used to classify and order East Asian ideographs, as found in the KangXi dictionary
UKannada	Enumerates names for characters in the Kannada script
UKatakana	Enumerates names for the Katakana syllabary used in writing Japanese

Class	Description
UKatakanaHiragana	Enumerates names for voicing marks used in both the Hiragana and Katakana syllabaries
ULao	Enumerates names for characters in the Lao script
ULatin	Enumerates names for characters in the Latin script
ULatin1	Enumerates names for Unicode characters from the ISO 8859-1 (Latin 1) character set
ULatinSmallLigature	Enumerates names for ligatures of lowercase Latin letters
ULetterLikeSymbol	Enumerates names for symbols, such as some mathematical constants, derived from ordinary letters from the Latin, Greek, or Hebrew scripts
UMalayalam	Enumerates names for characters in the Malayalam script
UMathematicalOperators	Enumerates names for symbols used in writing mathematics
UMiscellaneousTechnical	Enumerates names for miscellaneous technical symbols, such as keyboard symbols and crop marks
UModifierLetter	Enumerates names for modifier letters, such as most accents, used within Unicode
UOCR	Enumerates names for Unicode characters used in OCR systems such as check processing
UOriya	Enumerates names for characters in the Oriya script
UParenthesizedDigit	Enumerates names for digits inside parentheses
UParenthesizedHangul	Enumerates names for Hangul (Korean) characters inside parentheses
UParenthesizedIdeograph	Enumerates names for East Asian ideographs inside parentheses
UParenthesizedLatin	Enumerates names for Latin characters inside parentheses
UParenthesizedNumber	Enumerates names for numbers inside parentheses
UPresentationFormForVertical	Enumerates names for variants of Unicode characters used in writing text vertically
URomanNumeral	Enumerates names for Roman numeral characters
USmallVariants	Enumerates names for extra-small versions of certain Unicode characters
UStandardPhonetic	Enumerates names for Unicode characters used in the International Phonetic Alphabet
USubscript	Enumerates names for subscripted digits and letters

Class	Description
USuperscript	Enumerates names for superscripted digits and letters
UTamil	Enumerates names for characters in the Tamil script
UTelugu	Enumerates names for characters in the Telugu script
UThai	Enumerates names for characters in the Thai script
UUnicodeDigit	Enumerates names for Unicode characters used to write decimal numerals
UUnicodeSpecial	Enumerates names for special characters within Unicode, such as the byte order mark, null character, invalid character, and so on
UVulgarFraction	Enumerates names for fraction characters, such as 1/2
UZapfDingbats	Enumerates names for the Zapf dingbat set

RELATED CONCEPTS

Unicode Support and the IUnicode Class
Character Values
Character Properties

RELATED TASKS

Create a Unicode Application
"Identify a Character's Properties" on page 93
Identify a Character's Script
Find Characters with Specific Properties