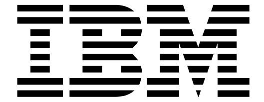
VisualAge® C++ Professional for AIX®



IBM® Open Class™: Error Handling, Tracing, and Testing

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.

© Copyright International Business Machines Corporation 1998, 2000. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices v
Programming Interface Information vii
Trademarks and Service Marks vii
Industry Standards viii

About This Book. ix

Chapter 1. Exceptions in the IBM Open Class 1
General Exceptions 2
  Signal Exceptions 3
  Use Throw Macros 3
  Use Assertion Macros 5
  Handle Exceptions 7
  Use try and catch 7
  Rethrow Exceptions 9
  Derive Your Own Exceptions 11

Chapter 2. Use Trace Macros 13

Chapter 3. Test Framework Components 17
The ITest Base Class 18
  Create a Base Test Class 18
  Set Up the Environment 19
  Test with Input Arguments 19
  Get Metainformation 20
  Example: A Simple Test 21
The Decision Function 22
  Run a Sequence of Tests 23
  Run Tests with Decision Functions. 23
Timing Tests 24
  Structure of Timing Loops 25
  Accuracy and Tolerance of Timing Tests 25
  Success Criteria and Precision of Timing Tests . . 26
  Example: Perform Timing Tests 28
Protocol Tests. 29
  Perform Protocol Tests. 30
Auxiliary Test Classes 30
  Print, Display, and Store Test Results 32
  Display Internal Test Information 33
Test Macros 34
  Example: Use Test Macros 35

© Copyright IBM Corp. 1998, 2000

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.

© Copyright IBM Corp. 1998, 2000

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.

© Copyright IBM Corp. 1998, 2000

Chapter 1. Exceptions in the IBM Open Class

Exceptions provide a mechanism for dealing with unusual circumstances in programs. Each exception object contains the following:

- Exception message text strings which describe the exception in detail.
- An error ID that lets you uniquely identify what error caused the exception.
- A severity code that lets you determine whether or not you can recover the exception.
- Information about where the exception was thrown.

Use the member functions of the exception classes to do the following:

- Add information about where the exception was thrown.
- Set, modify, and retrieve the exception text in the object.
- Get the error ID of the exception.
- Determine if the exception is recoverable.
- Log the exception data.
- Set the error ID of the exception.
- Set the severity of the exception.
- Set a trace function.

Using Exceptions
There are three primary ways to use the exception classes:

1. Catch exceptions that the IBM Open Class throws. Certain functions in the IBM Open Class throw exceptions that are derived from the IException class. If you are familiar with the characteristics of the exception classes, you can take advantage of this knowledge to make code that uses the IBM Open Class more robust.
2. Throw and catch your own exceptions. The exception classes provide a convenient way to package information about an exception.
3. Derive your own exception classes.

Open Class Error and Exception Output
Although Open Class is designed to catch as many errors as possible during the compilation and link steps, some errors can only be detected at run time. The classes in Open Class throw C++ exceptions to indicate runtime errors. Errors messages describing the exception can be seen while debugging, or can be seen in trace output sent to STDOUT, STDERR, or a queue; trace output is only seen if you have turned tracing on. Your own classes can also throw C++ exceptions and output trace information in the same way, by using classes provided in Open Class.

The Exceptions Message File
You can find IBM Open Class-provided exceptions in the following files:

-

cppaoi50.msg
-

© Copyright IBM Corp. 1998, 2000

ibmvaccl.cat

You must provide this file as part of you application's runtime in order to resolve exception text.

WIN

OS/2

You must rename the exception file if you ship your product on an Intel platform. To identify the new name, call IMessageText::setMessageFile(). See the topic Package and Distribute an IBM Open Class Application for more information.

RELATED TASKS

"Use Throw Macros" on page 3
"General Exceptions"
"Use Assertion Macros" on page 5
"Use try and catch" on page 7
"Rethrow Exceptions" on page 9
"Derive Your Own Exceptions" on page 11
"Chapter 2. Use Trace Macros" on page 13
Package and Distribute an IBM Open Class Application

General Exceptions

The IException class is the base exception class. When you want to signal that your code has encountered an unusual situation, you should throw whichever subclass of IException best conveys the nature of the situation. You can convey the nature of most unusual situations using one of the exceptions described in this section.

The following table lists some of the exception classes that are derived from IException and the situations in which they are typically thrown.

Exception Class	Thrown When...
IAccessError	A logical error occurs, such as "resource not found."
IAlreadyExists	Requested object could not be created because an object with the same name or location already exists.
IAssertionFailure	The expression in an IASSERT macro evaluates to false.
ICannotProceed	An operation could not be completed, but it may be possible to continue the operation or recover in some other fashion.
IConditionInvalid	An operation is attempted on an ICondition object that is in an invalid state.
IDecimalDataError	An IBinaryCodedDecimal or IDecimalUtil object encounters an underflow or overflow condition.
IDeviceError	A hardware-related error occurs.
IGraphicException	An exception occurs in 2D graphic code.
IInvalidName	Used an invalid name for an object (for example a file or network resource).
IInvalidParameter	An invalid parameter is passed; the expression in an IASSERTPARM macro evaluates to false.
IInvalidRequest	An object is in the wrong state for a function; the expression in an IASSERTSTATE macro evaluates to false.

Exception Class	Thrown When...
IMustBeEmpty	An operation on a container failed because the container was not empty.
IObjectNotFound	An operation failed because it was unable to locate the requested object.
IOutOfMemory	Memory is exhausted.
IOutOfSystemResource	System resource is exhausted.
IOutOfWindowResource	Window resource is exhausted.
IResourceExhausted	A resource is exhausted or currently unavailable.
IWaitAbandoned	A host abandons a wait before timing out

The following classes are used in conjunction with the IException class and its derived classes:

Class	Description
IException::TraceFn	A support class for the IException class.
IExceptionLocation	A support class for the IException class. The IEXCEPTION_LOCATION macro expands to create an instance of the IExceptionLocation class.

For more class-specific exceptions, refer to the following topics:
- File System Exceptions
- Exceptions Defined by the Streaming Classes
- Collection Class Library Exceptions

RELATED CONCEPTS

"Chapter 1. Exceptions in the IBM Open Class" on page 1
File System Exceptions
Exceptions Defined by the Streaming Classes
Collection Class Library Exceptions

RELATED TASKS

"Use Throw Macros"
"Use Assertion Macros" on page 5

Signal Exceptions

Use Throw Macros

When your code encounters an exceptional situation, you should identify which subclass of IException best conveys the nature of the situation and then throw that exception using an appropriate throw macro. There are nine throw macros:

Macro	Type of Exception Thrown
ITHROW	Any IBM Open Class exception. (See IException)
ITHROWCLIBERROR	Exceptions constructed with ICLibErrorInfo information.

Macro	Type of Exception Thrown
ITHROWERROR	Any IBM Open Class exception. (See IBaseErrorInfo.)
ITHROWERROR1	Any IBM Open Class exception. (See IBaseErrorInfo.)
ITHROWGUIERROR	Exceptions constructed with IGUIErrorInfo information.
ITHROWGUIERROR2	Exceptions constructed with IGUIErrorInfo information.
ITHROWLIBRARYERROR	Any IBM Open Class exceptions (see IBaseErrorInfo)
ITHROWLIBRARYERROR1	Any of the IBM Open Class exceptions (see IBaseErrorInfo)
ITHROWMMERROR	Exceptions constructed with IMMErrorInfo information.
ITHROWMMERROR2	Exceptions constructed with IMMErrorInfo information.
ITHROWSYSTEMERROR	Exceptions constructed with ISystemErrorInfo information.
ITHROWXLIBERROR	Exceptions constructed with IXLibErrorInfo information.

The INO_EXCEPTIONS_SUPPORT macro is provided in support of compilers that lack an exception handling implementation. If it is defined, the ITHROW macro ends the program after capturing the location information and logging it, instead of throwing an exception. This macro may not work correctly on all compilers.

The following macros end the program if you have defined INO_EXCEPTIONS_SUPPORT:

- ITHROW
- ITHROWERROR
- ITHROWERROR1
- ITHROWLIBRARYERROR
- ITHROWLIBRARYERROR1
- ITHROWGUIERROR
- ITHROWGUIERROR2
- ITHROWSYSTEMERROR

See IException for more information about the INO_EXCEPTIONS_SUPPORT macro.

The following example demonstrates the use of the ITHROW macro. The program attempts to output the first character of the file source.dat. If the program cannot locate the file, it throws an IAccessError object. The catch block then generates the error message “Cannot locate file.”

```
// Use the ITHROW macro
#include <iostream.h>
#include <fstream.h>
#include <iexcept.hpp>
void openFile(fstream& fs, char *filename)
{
   fs.open(filename, ios::in);
```

```
}
char getFirstChar(fstream& fs)
{
   char c;
   if (fs.get(c) == 0)
   {
      ITHROW(IAccessError("Cannot locate file."));
   }
   return c;
}
int main()
{
   char c;
   char *filename = "source.dat";
   fstream fs;
   openFile(fs, filename);
   try
   {
      c = getFirstChar(fs);
      cout << "Here is first character: "
           << c << endl;
   }
   catch(IAccessError ie)
   {
      cout << ie.text() << endl;
   }
   return 0;
}
```

RELATED CONCEPTS

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

RELATED TASKS

"General Exceptions" on page 2
"Use Assertion Macros"

Use Assertion Macros

The assertion macros provide a convenient mechanism for testing if a given expression is true and, if it is not true, automatically throwing an exception. When you use an assertion macro, you pass as input an expression that you anticipate to be true.

There are three assertion macros:

Assertion Macro	Description
IASSERT	This macro tests the expression you provide and, if it is false, throws the IAssertionFailure exception.
IASSERTSTATE	This macro tests the expression you provide and, if it is false, invokes the IExcept__assertState function. The IExcept__assertState function creates an IInvalidRequest exception. The macro adds location information, logs, and throws the exception.

Assertion Macro	Description
IASSERTPARM	This macro tests the expression you provide and, if it is false, invokes the IExcept__assertParameter function. The IExcept__assertParameter function creates an IInvalidParameter exception. The macro adds location information, logs, and throws the exception.

Consider the following simple example of using the IASSERTSTATE macro. The getFirstChar function contains the IASSERTSTATE macro and passes a call to the get function to the macro. If the get call fails, it returns zero and the IASSERTSTATE macro throws an IInvalidRequest exception.

```
// Using the IASSERTSTATE macro

#include <iostream.h>
#include <fstream.h>
#include <iexcept.hpp>

void openFile(fstream& fs, char *filename)
{
   fs.open(filename, ios::in);
}
char getFirstChar(fstream& fs)
{
   char c;
   IASSERTSTATE(fs.get(c));
   return c;
}

void main()
{
   char c;
   char * filename = "source.dat";
   fstream fs;
   openFile(fs, filename);
   try
   {
      c = getFirstChar(fs);
      cout << "Here is first character: " << c << endl;
   }
   catch(IException ie)
   {
      cout << "Type of exception is: " << ie.name() << endl;
      cout << "Location of exception is: "
           << ie.locationAtIndex(0)->fileName() << endl;
      if (ie.isRecoverable())
      {
         cout << "Exception is recoverable" << endl;
      }
      else
      {
         cout << "Exception is unrecoverable" << endl;
      }
   }
}
```

Suppose that you execute this example, and the source.dat file is not available. The call to open in the OpenFile function will fail. When the program calls getFirstChar within the try block, the IASSERTSTATE macro will throw an exception. The catch statement in main() will catch this exception, and the output will look something like this:

```
Type of exception is: IException
Location of exception is: iopen.cpp
Exception is recoverable
```

RELATED CONCEPTS

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

RELATED TASKS

"Use Throw Macros" on page 3
"General Exceptions" on page 2

Handle Exceptions

Use try and catch

Many functions throw exceptions when they encounter unusual situations. To ensure that your code is robust, you should call such functions from within a try-block and then either handle or rethrow any resulting exceptions from within a catch-statement.

Because all exceptions are derived from the IException class, a single catch statement can catch any exception that might be thrown. The following example illustrates how you can do this:

```
catch(IException &ie)
{
   // ...
   // code for all exception class exceptions
}
```

On the other hand, if you wanted to deal with each kind of exception separately, you could have catch statements that looked like this:

```
catch(IAccessError &ia)
{
   // ...
   // code for IAccessError exceptions
}
catch(IAssertionFailure &iaf)
{
   // ...
   // code for IAssertionFailure exceptions
}
// ...
```

The following example illustrates how you can safely call a function that might throw an exception. This sample code calls the new operator to create a huge array of integer pointers. The new operator throws an IOutOfMemory exception when there is insufficient memory available to satisfy a request for memory.

In the following code, a single invocation of the new operator exhausts all of the memory that is available for allocation. The catch statement specifies the base class IException rather than IOutOfMemory. If you know that a member function may throw an exception, but you do not know its exact type, you can specify a catch statement like this one to catch any possible exception.

```
// The new operator throwing an exception

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

```
#define TOOBIG 1000000000

int main()
{
   int i;
   try
   {
      int* istr = new int[TOOBIG];
   }
   catch(IException &ie)
   {
      std::cout << "Type of exception is: "
                << ie.name() << std::endl;
      std::cout << "Location of exception is: "
                << ie.locationAtIndex(0)->fileName()
                << std::endl;
      if (ie.isRecoverable())
         std::cout << "Exception is recoverable"
                   << std::endl;
      else
         std::cout << "Exception is unrecoverable"
                   << std::endl;
   }
   return 0;
}
```

Assuming that the constant TOOBIG is large enough to exhaust all of the memory available for allocation, this code produces the following output:

```
Type of exception is: IOutOfMemory
Location of exception is: ibase.cpp
Exception is unrecoverable
```

The following is another example illustrating how you can safely call a function that might throw an exception. If you use the subscript operator on an IString object that is declared const, the operator will throw an IInvalidRequest exception if the index is out of the bounds of the IString object.

In the following code, an IString object is declared const, and then the subscript operator is used with an index beyond the size of the object.

```
// Example that causes a subscript
// out of bounds exception

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

int main()
{
   try
   {
      const IString ConstStr = "OFF";
      std::cout << ConstStr[4] << std::endl;
   }
   catch(IException &ie)
   {
      std::cout << "Type of exception is: "
                << ie.name() << std::endl;
      std::cout << "Location of exception is: "
                << ie.locationAtIndex(0)->fileName()
                << std::endl;
      if (ie.isRecoverable())
      std::cout << "Exception is recoverable"
                << std::endl;
      else
         std::cout << "Exception is unrecoverable"
```

```
                    << std::endl;
    }
    return 0;
}
```

Because the index is beyond the size of the IString object, the subscript operator throws an exception. When this code is run, the following output is produced:

```
Type of exception is: IInvalidRequest
Location of exception is: istring5.cpp
Exception is recoverable
```

User Interface Classes Exceptions

The User Interface Classes have a special mechanism for catching exceptions thrown from event handler objects. To catch those exceptions, follow these steps:

1. Create a class derived from IWindow::ExceptionFn.
2. Override its virtual handleException function.
3. Create an object of this derived class.
4. Pass this object to the setExceptionFunction static function of IWindow::ExceptionFn.

The IWindow class will call its handleException function to process any exceptions thrown during event handling.

RELATED CONCEPTS

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

RELATED TASKS

"Rethrow Exceptions"
"Derive Your Own Exceptions" on page 11

Rethrow Exceptions

Once you catch an exception that has been thrown, you can rethrow it using the IRETHROW macro.

The IRETHROW macro accepts as input a predefined instance of any subclass of IException that has been previously thrown and caught. Like the ITHROW macro, it also captures the location information, and logs all instance data before rethrowing the exception.

The following example demonstrates the use of the IRETHROW macro:

1. The program attempts to output the first character of the file source.dat.
2. If this attempt fails, the program throws an exception, then tries to access the file source.bak.
3. If the program fails to access the file source.bak, the program rethrows the exception.

```
// Using the IRETHROW macro
#include <iostream.h>
#include <fstream.h>
#include <iexcept.hpp>
void openFile(fstream& fs, char *filename)
{
  fs.open(filename, ios::in);
}
char getFirstChar(fstream& fs)
{
```

```
  char c;
  if (fs.get(c) == 0)
  {
    ITHROW(IAccessError("Cannot locate file."));
  }
  return c;
}
char accessSourceFile()
{
  fstream fs;
  char c;
  char *filename = "source.dat";
  cout << "Accessing file: " << filename << endl;
  openFile(fs, filename);
  try
  {
    return getFirstChar(fs);
  }
  catch(IAccessError &ie)
  {
    cout << ie.text() << endl;
    cout << "Trying source.bak" << endl;
    fstream fsBackup;
    char *backup = "source.bak";
    openFile(fsBackup, backup);
    if (fs.get(c) == 0)
    {
      IRETHROW(ie);
    }
    else
    {
      return c;
    }
  }
}
int main()
{
  char c;
  try
  {
    c = accessSourceFile();
    cout << "Here is first character: "
         << c << endl;
  }
  catch(IAccessError &ie)
  {
    cout << ie.text() << endl;
  }
  return 0;
}
```

If the program cannot access either of the source files (source.dat and source.bak), the program outputs the following:

```
Accessing file: source.dat
Cannot locate file.
Trying source.bak
Cannot locate file.
```

RELATED CONCEPTS

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

RELATED TASKS

“Use try and catch” on page 7
“Derive Your Own Exceptions”

Derive Your Own Exceptions

When deriving your own exception classes from IException, if you do not need to add more functionality to your classes, such as specialized member functions or variables, you can use the IEXCLASSDECLARE and IEXCLASSIMPLEMENT macros:

- The IEXCLASSDECLARE macro creates a declaration for a subclass of IException or one of its subclasses.
- The IEXCLASSIMPLEMENT creates a definition for a subclass of IException or one of its subclasses.

These macros allow you to easily create a hierarchy of exception classes. The following example demonstrates the use of the IEXCLASSDECLARE and IEXCLASSIMPLEMENT macros. The program creates a simple hierarchy of exceptions. These exceptions have the same functionality as IAccessError exceptions. The main() function demonstrates an advantage of using a hierarchical structure for your exceptions: the ability to more precisely define and handle anomalies that might occur in your program:

```
// Using the IEXCLASSDECLARE and IEXCLASSIMPLEMENT macros
#include <iostream.h>
#include <fstream.h>
#include <iexcept.hpp>
IEXCLASSDECLARE (MyFileError, IAccessError);
IEXCLASSDECLARE (MyCannotOpenFile, MyFileError);
IEXCLASSDECLARE (MyCannotSaveFile, MyFileError);
IEXCLASSIMPLEMENT (MyFileError, IAccessError);
IEXCLASSIMPLEMENT (MyCannotOpenFile, MyFileError);
IEXCLASSIMPLEMENT (MyCannotSaveFile, MyFileError);
int main()
{
  try
  {
    ITHROW(MyCannotSaveFile("My derived exception."));
  }
  catch(MyCannotOpenFile &ie)
  {
    cout << "Catch block that accepts "
         << "MyCannotOpenFile exceptions." << endl
         << "Type of exception is: " << ie.name() << endl;
  }
  catch(MyCannotSaveFile &ie)
  {
    cout << "Catch block that accepts "
         << MyCannotSaveFile exceptions." << endl
         << "Type of exception is: " << ie.name() << endl;
  }
  catch(IException &ie)
  {
     cout << "Catch block that accepts all "
          << "IException derived exceptions." << endl
          << "Type of exception is: " << ie.name() << endl;
  }
  return 0;
}
```

This program produces the following output:

```
Catch block that accepts MyCannotSaveFile exceptions.
Type of exception is: MyCannotSaveFile
```

RELATED CONCEPTS

"Chapter 1. Exceptions in the IBM Open Class" on page 1
"General Exceptions" on page 2

RELATED TASKS

"Use Throw Macros" on page 3
"General Exceptions" on page 2
"Use Assertion Macros" on page 5
"Use try and catch" on page 7
"Rethrow Exceptions" on page 9
"Chapter 2. Use Trace Macros" on page 13

Chapter 2. Use Trace Macros

The ITrace class provides a set of facilities that allow you to put trace statements in your code. When an exception is thrown, trace records are output with information about the exception. The ITrace class allows you to send trace output to standard output or to capture it in a file.

By defining certain macros, you can selectively turn tracing on and off. There are three special trace macros:

- IC_TRACE_RUNTIME
- IC_TRACE_DEVELOP
- IC_TRACE_ALL

By defining or not defining these macros, you can specify whether or not the trace macros are expanded, and thus whether or not your program produces trace output.

If IC_TRACE_RUNTIME is defined, the following macros are expanded:

Macro	Description
IMODTRACE_RUNTIME	Takes one argument that is the name of the current module. It creates an ITrace object using the module name as the name of the trace and the current line number as the line number. You cannot pass a temporary string to this macro. In most cases you can use IFUNCTRACE_RUNTIME instead.
IFUNCTRACE_RUNTIME	This macro takes no arguments. It creates an ITrace object using the function name as the name of the trace and the current line number as the line number.
ITRACE_RUNTIME	This macro takes a single argument. This argument is written to the trace location.

If IC_TRACE_DEVELOP is defined, all of the macros that are expanded when IC_TRACE_RUNTIME is defined are also expanded. In addition, the following macros are expanded:

Macro	Description
IMODTRACE_DEVELOP	Rakes one argument. Typically you use the argument to name the current module. This macro creates an ITrace object using the module name as the name of the trace and the current line number as the line number. You cannot pass a temporary string to this macro. In most cases you can use IFUNCTRACE_DEVELOP instead.
IFUNCTRACE_DEVELOP	Takes no arguments. It creates an ITrace object using the function name as the name of the trace and the current line number as the line number.

© Copyright IBM Corp. 1998, 2000

Macro	Description
ITRACE_DEVELOP	Takes a single argument. This argument is written to the trace location.

If IC_TRACE_ALL is defined, all of the macros that are expanded when IC_TRACE_DEVELOP is defined are also expanded. In addition, the following macros are expanded:
- IMODTRACE_ALL
- IFUNCTRACE_ALL
- ITRACE_ALL

The ALL prefix will define all trace macros.

The following code shows one way that you could use the trace macros to produce trace output for your programs. In this code, the macros IFUNCTRACE_DEVELOP and ITRACE_DEVELOP are used to create trace statements that indicate that the flow of control has passed through the functions openFile and getFirstChar.

```
// Producing trace output with the ITrace class

#define IC_TRACE_DEVELOP
#include <iostream>
#include <fstream>
#include <iexcept.hpp>
#include <itrace.hpp>
void openFile(std::fstream& fs, char *filename)
{
   IFUNCTRACE_DEVELOP();
   std::fs.open(filename, std::ios::in);
   ITRACE_DEVELOP("after open statement");
}
char getFirstChar(std::fstream& fs)
{
   char c;
   IFUNCTRACE_DEVELOP();
   std::fs.get(c);
   ITRACE_DEVELOP("after get statement");
   return c;
}
int main()
{
   char c;
   char * filename = "source.dat";
   std::fstream fs;
   // static functions to enable tracing and direct
   // tracing output to standard output
   ITrace::enableTrace();
   ITrace::writeToStandardOutput();
   openFile(fs, filename);
   c = getFirstChar(fs);
   std::cout << "Here is first character: "
             << c << std::endl;
   return 0;
 }
```

Notice that, in this code, the static functions enableTrace and writeToStandardOutput are used to enable tracing and to direct the trace output to standard output.

Because the macro IC_TRACE_DEVELOP is defined, the trace macros produce trace output. In addition, the trace output has been explicitly directed to standard output, so the output of the code looks like this:

```
+openFile(fstream&,char*)
  >after open statement
-openFile(fstream&,char*)
+getFirstChar(fstream&)
  >after get statement
-getFirstChar(fstream&)
Here is first character: t
```

Suppose that you wanted to turn off the trace output in this program. One way to do it is to modify the code so that the macro IC_TRACE_DEVELOP is not defined. If you do this, the trace macros are not expanded, and no trace output is produced. The output of this code with IC_TRACE_DEVELOP not defined looks like this:

```
Here is first character: t
```

Trace Environment Variables
In order to obtain runtime trace information from IBM Open Class, you must set the environment variables ICLUI_TRACE and ICLUI_TRACETO. The ICLUI_TRACETO environment variable takes precedence over the ICLUI_TRACE environment variable. To turn off tracing, unset the ICLUI_TRACETO variable unset or set to OFF the ICLUI_TRACE environment variable.

The folowing table describes the trace environment variables:

Trace Environment Variables	Description
ICLUI_TRACE	Can be set to the following: • OFF (default value) • ON • NOPREFIX (no prefix information is written)
ICLUI_TRACETO	Can be set to the following: • STDERR or ERR (writes to standard error) • STDOUT or OUT (writes to standart out) • FILE (writes to a file) • QUEUE (writes to queue. This is the default value.) • PMPRINTF AIX Standard output and queue are the same.
ICLUI_TRACEFILE	Specifies the file name. This is used when ICLUI_TRACETO is set to FILE.
ICLUI_CHECKSTACK	WIN OS/2 Can be set to ON or TRUE to check the renaming stack size.

RELATED CONCEPTS

"Chapter 1. Exceptions in the IBM Open Class" on page 1
Application Testing Overview

RELATED TASKS

"Use Throw Macros" on page 3
"General Exceptions" on page 2
"Use Assertion Macros" on page 5
"Use try and catch" on page 7
"Rethrow Exceptions" on page 9
"Derive Your Own Exceptions" on page 11

Chapter 3. Test Framework Components

The core of the Test Framework consists of the base class ITest and a number of associated classes, which can be subdivided into several categories according to their characteristics:

- Multiple tests:
  - ITestCollection
  - ITestMultiplexer
- Timing tests:
  - ITimingTest
  - IStartStopTimingTest
- Protocol tests
  - IComparisonTestOf
  - IStreamTestOf
- Auxiliary classes
  - ITieredTextBuffer
  - IArgumentDictionary
- Test macros:
  - for single tests:
    - runTestImplementationMacro
    - beginTestFrameworkMacro
  - for several tests:
    - runTestMacro
    - endTestFrameworkMacro
    - runTestResultMacro
  - for protocol tests:
    - comparisonTestMacro
    - IStreamTestMacro

Test Classes

A *target class* is a class in your program that you want to test for correct performance. You write a *test class*, derived from a Test Framework class, that runs the target class and does the evaluation.

ITest is the abstract base class from which all test classes are derived, including the test class that you write and declare to be a friend of your target class. You can derive directly from ITest to perform simple tests. In addition specialized subclasses derived from ITest provide three kinds of functionality:

- *Multiple tests* group tests so that you can perform more than one test at a time
- *Timing tests* performs timing measurements
- *Protocol tests* traverse an entire tree of classes and tests them adhering to the same protocol

The test classes work by creating and evaluating an instance of your target class. You do not have to make changes directly to your target class code to test its behavior.

© Copyright IBM Corp. 1998, 2000

Runtest macros
The Test Framework defines several macros that you may use to create applications to run your tests:

- To automatically generate a main function to run a single test, use runTestImplementationMacro.
- To create your own application to run several tests, use beginTestFrameworkMacro, runTestMacro, endTestFrameworkMacro, and runTestResultMacro.
- For protocol tests, use comparisonTestMacro and IStreamTestMacro.

RELATED CONCEPTS

The ITest Base Class

The core of the Test Framework is the class ITest, an abstract base class that defines the protocol for an executable test. ITest is a framework that contains the following:

- A test success state (not set, pass, or fail)
- A target object on which tests are performed
- Input through a collection of IString objects
- Output via an object with ostream-like behavior
- Textual metainformation description about the test such as the purpose of the test and the name of the class being tested
- Uniform exception handling by the Test Framework

To write a test, you generally write a subclass of ITest. This subclass overrides the test function and, optionally, the associated framework functions reset, setup, and cleanup. To run the test, you generally use the runTestImplementationMacro to create an application that defines the main function to create and run the test.

RELATED CONCEPTS

Application Testing Overview
"Chapter 3. Test Framework Components" on page 17
"Test Macros" on page 34

RELATED TASKS

"Create a Base Test Class"
"Set Up the Environment" on page 19

Create a Base Test Class

Follow these steps to create a test:

1. Identify the types of operations you want to test.
2. Derive your test class from the appropriate Test Framework class. For most tests, you derive your test from ITest, so the minimum requirement for using the Test Framework is to include `itest.hpp`.
3. Give your test class access to the target class you are testing.
4. Override the test function in your test class.
5. Override setup and cleanup functions, if necessary.
6. Override the inherited ITest functions with your data members. The functions you need to override are operator<<= and operator>>=.
7. When you are ready to perform the test, use a runtime macro.

RELATED CONCEPTS

"Chapter 1. Exceptions in the IBM Open Class" on page 1
Application Testing Overview

RELATED TASKS

"Set Up the Environment"
"Test with Input Arguments"
"Get Metainformation" on page 20
"Example: A Simple Test" on page 21

Set Up the Environment

The ITest::setup() function allows you to prepare for your test before calling the ITest::test() function. Override setup to specify any special conditions that the test requires. A common activity to handle during setup is to parse command-line arguments that you can pass into a test.

The Test Framework provides the cleanup function, which you can use to perform any necessary cleanup after your test finishes. You need to override cleanup to restore the system to its previous state by undoing any actions, such as storage allocation, performed within setup that affect the state of the system. The cleanup function is executed even if a software exception occurs. If a hardware exception occurs in the ITest function, such as a bus error, then cleanup is not called.

RELATED CONCEPTS

"Chapter 1. Exceptions in the IBM Open Class" on page 1
Application Testing Overview

RELATED TASKS

"Create a Base Test Class" on page 18
"Test with Input Arguments"
"Get Metainformation" on page 20
"Example: A Simple Test" on page 21

Test with Input Arguments

This example shows how a test can read input arguments.

```
#include <itest.hpp>
#include <iargdict.hpp>
class INumberATTest : public ITest {
public:
                     INumberATTest();
   virtual          ~INumberATTest();
protected:
    virtual void    setup();
    virtual void    test();
private:
    long            fTimingCount;
    long            fSampleCount;
};
 INumberAtTest::INumberAtTest()
{
    fTimingCount = 5; // Default value
    fSampleCount = 10; // Default value
}
```

```
INumberAtTest::~INumberAtTest() { }
void INumberAtTest::test()
{
    outputTextStream() << "inumberattest timing count="
        << fTimingCount << " , sample count="
        << fSampleCount << '\n';
    SetSuccess(true);
}
void INumberAtTest::setup()
/*
    Setup reads input arguments of the form:
        [-c <timing count>] [-s <sample count>]
    The timing count must be in the range 1..256. The sample count must
    be greater than 5. If the counts are not given, they remain unchanged.
    If a bad input is given, the test fails.
*/
{
    static const IString kTimingCountKey(" c");
    const long kmintimingcount="1;"
    const long kmaxtimingcount="256;"
    static const istring ksamplecountkey("-s");
    const long kminsamplecount="5;"
    iargumentdictionary args(*this);

    if (!args.numberat(ktimingcountkey, ftimingcount, kmintimingcount, kmaxtimingcount) ||
         !args.numberat(ksamplecountkey, fsamplecount, kminsamplecount)) {

       outputtextstream() << "error parsing input arguments.\n";
       setsuccess(false);
    }
}
```

RELATED CONCEPTS

"Chapter 1. Exceptions in the IBM Open Class" on page 1
Application Testing Overview

RELATED TASKS

"Create a Base Test Class" on page 18
"Set Up the Environment" on page 19
"Get Metainformation"
"Example: A Simple Test" on page 21

Get Metainformation

The Test Framework provides a means for getting textual metainformation about a test. To do this, override the ITest::copyInfo() function. Inside copyInfo, call the inherited implementation, then add key-value pairs with the metainformation for their ITest subclass.

For example:

```
class IMyTest: public ITest
{
    // ...
protected:
    void copyInfo();
};
void IMyTest::copyInfo()
{
   ITest::copyInfo();
   addInfo(kDescriptionKey, "Test which tests ITest.");
   addInfo(kInputSyntaxKey, "[-r <run count>]");
   addInfo(kTargetClassKey, "ITarget");
}
```

RELATED CONCEPTS

"Chapter 1. Exceptions in the IBM Open Class" on page 1
Application Testing Overview

RELATED TASKS

"Create a Base Test Class" on page 18
"Set Up the Environment" on page 19
"Test with Input Arguments" on page 19
"Example: A Simple Test"

Example: A Simple Test

The ITest subclass `ISimpleTest` is a very simple test of `ITarget`. `ISimpleTest` overrides ITest::test() to check if `ITarget::maxLength()` returns a correct value. To perform this test, it creates an object of a target class and calls `maxLength()`. If the result is correct, it sets Success to true; otherwise, it sets Success to false.

The following is the listing of the file itarget.hpp:

```
// itarget.hpp
class ITarget
{
public:
   ITarget(int maxlength) { fMaxLength = maxlength;}
   virtual ~ITarget() {}
   int maxLength() { return fMaxlength; }
private:
    int fMaxlength;
};
```

The following is the listing of the file itarget.cpp:

```
// itarget.cpp
// test class
#include <runtest.hpp>
#include <itest.hpp>
#include "itarget.hpp"
class ISimpleTest : public ITest
{
public:
   ISimpleTest();
   virtual ~ISimpleTest();
protected:
    virtual void test();
private:
};
ISimpleTest::ISimpleTest() { }
ISimpleTest::~ISimpleTest() { }
void ISimpleTest::test()
{
   int correctLength = 100;
   ITarget target(correctLength);
   if (target.maxLength() != correctLength)
   {
      outputTextStream() << "length doesn't match";
      setsuccess(false);
   }
   else
      setsuccess(true);
}
runtestimplementationmacro(isimpletest);
```

RELATED CONCEPTS

"Chapter 1. Exceptions in the IBM Open Class" on page 1
Application Testing Overview

RELATED TASKS

"Create a Base Test Class" on page 18
"Set Up the Environment" on page 19
"Test with Input Arguments" on page 19
"Get Metainformation" on page 20

The Decision Function

You can group your tests by combining multiple operations into a single test class.

A *decision function* is the code you write that determines if the code you are testing does what you expect. Using a single ITest object to test several functions allows the test decision functions to share code and functions.

The ITestCollection Class
ITestCollection is a subclass of ITest that tests a collection of ITest objects. This allows you to group related ITest subclasses together within a single test. The tests in the collection are run sequentially to determine the success or failure of the entire group. The resulting test passes if all of its subtests pass; otherwise fails. By default, the subtests are always executed in a fixed order (the order in which you add subtests). The subtests can be run in random order if you call the randomlyReorder function after adding subtests. By default, ITestCollection does not halt if a subtest fails, but runs all the subtests. To change this default behavior, call setHaltOnFail(true) in your setupSubtests.

The ITestMultiplexer Class
ITestMultiplexer is a subclass of ITest that supports the model of multiple decision functions applied to a single test target. These decision functions are invoked using test keys. You write a group of decision functions, then build a table that maps keys to functions. This helps you avoid the duplication of setup and cleanup code for the functions that share them.

Keys are case-sensitive IString objects that select the decision function to run when the test function is called. You can use keys in three ways:

- Specify a single key with setDecisionKey.
- Specify the key kAllDecisions with setDecisionKey. This will call all decision keys.
- Specify how many of the ITest input arguments will be interpreted as keys with the setInputsAsKeys function.

The default behavior is to run all decisions.

RELATED CONCEPTS

Application Testing Overview
"Chapter 3. Test Framework Components" on page 17

RELATED TASKS

"Run a Sequence of Tests" on page 23
"Run Tests with Decision Functions" on page 23

Run a Sequence of Tests

IColTest is a subclass of ITestCollection. It tests three subtests (`ISubtest1`, `ISubtest2`, and `ISubtest3`), which are all subclasses of ITest. The example runs the tests in random order. Also, `IColTest` halts on the failure of any subtest.

```
class IColTest : public ITestCollection
{
public:
IColTest();
˜IColTest();
void setupSubtests();
};
IColTest::IColTest() {}
IColTest::˜IColTest() {}
void IColTest::setupSubtests
{
ISubtest1* test1 = new ISubtest1(100);
ISubtest2* test2 = new ISubtest2();
ISubtest3* test3 = new ISubtest3("Hello");
adoptTest(test1);
adoptTest(test2);
adoptTest(test3);
randomlyReorder(56734);
setHaltOnFail(true);
}
runTestImplementationMacro(IColTest);
```

RELATED CONCEPTS

Application Testing Overview
"Chapter 3. Test Framework Components" on page 17
"The Decision Function" on page 22

RELATED TASKS

"Run Tests with Decision Functions"

Run Tests with Decision Functions

Run tests with decision functions using ITestMultiplexer.

In the following example, `IMulTest` is a subclass of ITestMultiplexer. There are three decision functions, `appleFunction`, `bearFunction`, and `cartFunction`. Each of these functions tests the different aspects of the target class (although they just return true in this example). Decision functions should accept no parameter and should return bool. `IMulTest` reports PASS when all the three functions return true:

```
class IMulTest: public ITestMultiplexer
{
public:
IMulTest();
˜IMulTest();
void loadDecisions();
bool appleFunction();
bool bearFunction();
bool cartFunction();
};
IMulTest::IMulTest() {}
IMulTest::˜IMulTest() {}
void IMulTest::loadDecisions()
{
addDecision("apple",( ITestDecisionFn)appleFunction);
addDecision("bear", (ITestDecisionFn)bearFunction);
addDecision("cart", (ITestDecisionFn)cartFunction);
```

```
}
bool IMulTest::appleFunction() { return true;}
bool IMulTest::bearFunction() {return true;}
bool IMulTest::cartFunction() {return true;}
runTestImplementationMacro(IMulTest);
```

RELATED CONCEPTS

Application Testing Overview
"Chapter 3. Test Framework Components" on page 17
"The Decision Function" on page 22

RELATED TASKS

"Run a Sequence of Tests" on page 23

Timing Tests

The ITimingTest base class enables you to write tests that measure the time a specific operation takes to complete. The derived class IStartStopTimingTest gives you more control in timing events.

ITimingTest is the principle class used to measure CPU performance. It measures the time a well-defined operation (such as adding a member to an IACollection subclass) takes to execute.

ITimingTest expects the operation to show some variation in execution time because of multitasking, varying CPU load, and other variable conditions. Consequently, the class takes multiple measurements and reports the median.

ITimingTest contains three framework functions: TimingSetup, TimingTest, and TimingCleanup. Only the TimingTest function is timed. The TimingSetup and TimingCleanup functions run before and after TimingTest, but are not themselves timed.

IStartStopTimingTest
IStartStopTimingTest is an abstract framework class for timing measurements too complicated to do with ITimingTest. It is a subclass of ITimingTest that has the following characteristics:

1. Lower precision
2. The ability to start and stop the timer

Use IStartStopTimingTest if you have to start and stop the timer multiple times. Because of the lower precision, use ITimingTest whenever possible.
To use, subclass and override startStopTimingTest in the same manner as you would override ITimingTest::timingTest.

For example, consider a class *INetThing* that queries the network. Only the first instantiation of *INetThing* performs a certain action. To test the initial object-constructor time of *INetThing*, you would delete each existing *INetThing* before the next one is constructed. You would use the following sequence within the timing loop:

1. Begin timing.
2. Instantiate *INetThing*.
3. Stop timing.
4. Delete *INetThing*.

Because it takes a substantial amount of time to measure time itself, when the timer is started or stopped, IStartStopTimingTest can only measure times in milliseconds.

RELATED CONCEPTS

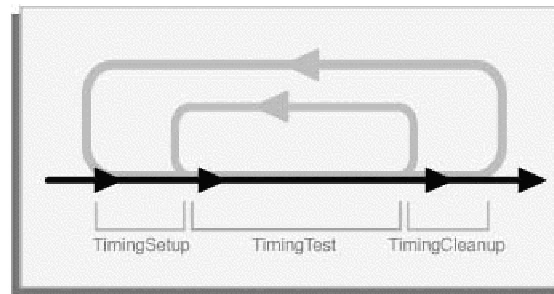
Application Testing Overview
"Chapter 3. Test Framework Components" on page 17
"Structure of Timing Loops"
"Accuracy and Tolerance of Timing Tests"
"Success Criteria and Precision of Timing Tests" on page 26

RELATED TASKS

"Example: Perform Timing Tests" on page 28

Structure of Timing Loops

The time needed to execute the timingTest function is measured by taking a number of samples called the *sample count*. Each sample consists of a call to timingSetup, a number of calls to timingTest, and a call to timingCleanup.



The calls to timingTest happen in a tight loop called the *timing loop*. Subclasses or the caller can control how many iterations comprise the timing loop by changing the *timing count*. The total time taken by the timing loop is divided by the number of iterations to get a single *sample time*. The median of all sample times is taken as the estimate of the time to execute the TimingTest function.

RELATED CONCEPTS

Timing Tests
"Accuracy and Tolerance of Timing Tests"
"Success Criteria and Precision of Timing Tests" on page 26

RELATED TASKS

"Example: Perform Timing Tests" on page 28

Accuracy and Tolerance of Timing Tests

Accuracy

Calls to the timingTest function incur an overhead because of the time required to make the virtual function call. ITimingTest compensates for this by computing an

empty function time, which is the estimated time to make a call to an empty virtual function. This time is subtracted from each sample time in order to calibrate for this overhead.

Another way to increase the accuracy of timing tests is to increase the timing count to divide the various sources of error across more iterations. In general, the value of the timing count should be at least 100.

Tolerance
ITimingTest usually runs in *automatic mode*. In this mode, samples are taken sequentially until the error for the estimated time declines to some percentage. This percentage is the *tolerance*. For example, if the tolerance is 0.05, then samples will continue to be taken until the time measurement has an error of plus or minus 5%.

Automatic sampling estimates the mean of a moving window of samples. This window is *sample count* samples wide. The error of the mean is half of the 95% confidence interval. When this error (expressed as a fraction) declines below the tolerance, then the window is accepted and its mean returned as the estimated time.

It is possible that the time being measured might never stabilize, and that the required tolerance might never be reached. To handle this contingency, an upper limit has been set on the number of samples taken during automatic sampling. If the maximum number of samples is taken, and samples still have not converged, a warning is issued and the final window is accepted.

You might not want to use automatic timing for some tests. If you disable automatic timing, then the number of samples specified by the sample count is taken and their median returned as the measured time, regardless of the error. In this case, tolerance is ignored.

RELATED CONCEPTS

Timing Tests
"Structure of Timing Loops" on page 25
"Success Criteria and Precision of Timing Tests"

RELATED TASKS

"Example: Perform Timing Tests" on page 28

Success Criteria and Precision of Timing Tests

Success and Failure
A timing test fails:

- If some function calls `setSuccess(false)`
- If the time measurement itself fails
- If the test results in a value less than zero, which indicates a problem with calibration

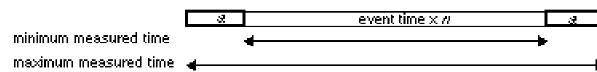
In the absence of these events, if the measurement reaches the required tolerance within the specified maximum allowed time, the test succeeds.

Precision
The timing test class attempts to correctly report the precision of its results by taking the following measurements:

- The standard deviation of all samples. This is used to compute a 95% confidence interval for the average.
- Timing overhead. This is measured by calling the internal timer repeatedly and measuring the time required to do so.

The contribution of timing overhead to errors in event-time measurement is estimated as follows:

The bar represents a single sample. The two boxes labeled *a* at the ends of the bar represent the timing overhead. The left end of the box is the time at which the call to a timing function is initiated, and the right end of the box is the time at which the call returns. The resulting value may represent time at any point within the box. The minimum measured time is exactly the event time multiplied by the number of iterations. The maximum measured time is the event time multiplied by the number of iterations plus 2*a*. This is the worst case. The actual event time falls somewhere in between.



ITimingTest reports the most pessimistic interpretation that is reasonable. It reports each timing measurement as the upper end of the 95% confidence interval for the event time, plus an upper-bound estimate of 2*a/n* for the timing overhead. A greater sample count decreases error by reducing the size of the confidence interval; a larger timing count decreases error by distributing the timing overhead over more events.

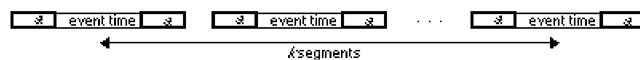
ITimingTest parses command line arguments. These arguments modify the way in which timing is done. They may also be supplied programmatically (see ITest).

Argument	Description
-at tol	Set the tolerance for automatic sampling to tol (default = 0.1).
-c count	Set the operations per sample to count (default = 100).
-cc	Calibrate continually by measuring the empty loop time before each sample.
-co correlation	Set the minimum correlation between samples for which to post an error to correlation (default = 0.4).
-m maxTime	Consider this test to fail if the median time per operation is greater than maxTime (default = none). The units of maxTime are in microseconds.
-na	Turn off automatic timing.
-s n	Take n samples, each sample consisting of many operations (default = 10).
-t	Do not do timing tests; just execute one operation without timing.
-w maxWindows	Stop if times do not converge within maxWindows of automatic sampling (default = 10).

The start-stop timing test attempts to correctly report the precision of its results by estimating the dispersion of samples and the sampling error as follows:

- Each sample consists of one or more segments. These segments represent timed events, beginning with a call to start an internal timer, the activity being timed, and a call to stop the internal timer. Let k be the number of segments per sample.
- The Test Framework makes the pessimistic assumption that the measured time for each segment has a potential error range of $2a$, where a is the timing overhead.

The contribution of timing overhead to errors in IStartStopTimingTest event-time measurement is also estimated. The boxes labeled a at the ends of the bar represent the timing overhead. This diagram represents a single sample, which consists of one or more segments. Each segment is bracketed on each end by the timing overhead.



- The minimum measured time is exactly the event time multiplied by the number of segments, k. The maximum measured time is the event time times k plus $2ka$. Thus, the worst-case error is $2ka$.
- As is the case with ITimingTest, increasing the sample count reduces the dispersion between samples. Unlike ITimingTest, however, increasing the timing count does nothing to reduce the sampling error because the timing overhead is incurred repeatedly within the sampling loop, rather than only at the start and finish.

RELATED CONCEPTS

Timing Tests
"Structure of Timing Loops" on page 25
"Accuracy and Tolerance of Timing Tests" on page 25

RELATED TASKS

"Example: Perform Timing Tests"

Example: Perform Timing Tests

This code shows a simple test that reports how long it takes to add an object to a sequence.

```
class ISequenceAddTimingTest : public ITimingTest {
public:
   ISequenceAddTimingTest();
   virtual ~ISequenceAddTimingTest();
protected:
   virtual void timingSetup();
   virtual void timingTest();
   virtual void timingCleanup();
private:
   IString **fObject;            // Array of pointers to objects to be added
   long fObjectArraySize;   // Number of items in fObject
   IString **fNextObject;        // Pointer to next pointer to add
   ISequence<IString> fSequence; // Deque to add to
};
ISequenceAddTimingTest::ISequenceAddTimingTest()
{
   fObject = NULL;
   fObjectArraySize = 0;
```

```
}
ISequenceAddTimingTest::~ISequenceAddTimingTest()
{
   timingCleanup(); // Just in case timingSetup was called without timingCleanup
}
void ISequenceAddTimingTest::timingSetup()
{
   timingCleanup(); // Just in case timingSetup was called without timingCleanup
   fObjectArraySize = timingCount();
   fObject = new IString*[fObjectArraySize];
   for (long i=0; i < fObjectArraySize; i++)
      fObject[i] = new IString;
   fNextObject = &fObject[0];
}
void ISequenceAddTimingTest::timingTest()
{
   fSequence.add(**fNextObject++);
}
void ISequenceAddTimingTest::timingCleanup()
{
   if (fObject != NULL) {
      for (long i=0; i < fObjectArraySize; i++)
      delete fobject[i];
      delete[] fobject;
      fObject= NULL;
   }
}
RunTestImplementationMacro(ISequenceAddTimingTest);
```

RELATED CONCEPTS

Application Testing Overview
Timing Tests
"Structure of Timing Loops" on page 25
"Accuracy and Tolerance of Timing Tests" on page 25
"Success Criteria and Precision of Timing Tests" on page 26

RELATED TASKS

"Create a Base Test Class" on page 18
"Set Up the Environment" on page 19

Protocol Tests

The Test Framework contains two ready-made protocol tests implemented using template classes:

- IComparisonTestOf
- IStreamTestOf

Protocol tests help leverage testing effort by allowing you to test classes when those classes are expected to adhere to some protocol.

If your object uses the default comparator and default streamer, protocol macros let you write entire protocol tests with a single line.

You can use the protocol macros whenever you can create the target with a single parameter list. You cannot use the macros if you need to create or reference other objects in order to create your target. In this case, you must derive your protocol test class from the appropriate template class.

RELATED CONCEPTS

Application Testing Overview
"Chapter 3. Test Framework Components" on page 17

RELATED TASKS

"Perform Protocol Tests"

Perform Protocol Tests

The global functions createComparisonTest and createStreamTest are useful if you want to create instances of IComparisonTestOf or IStreamTestOf and run those tests under ITestCollection using ITestCollection::adoptMethod.

```
class IProtocolTest : public ITestCollection
{
public:
IProtocolTest();
~IProtocolTest();
virtual void setupSubtests();
};
IProtocolTest::IProtocolTest() {}
IProtocolTest::~IProtocolTest() {}
void IProtocolTest::setupSubtests()
{
adoptTest(createComparisonTest(new IMyTest(1), new IMyTest(2)));
adoptTest(createStreamTest(new IMyTest(1)));
adoptTest(createComparisonTest(new IMyTest2MyTest2("abc"), new IMyTest2("xyz")));
adoptTest(createStreamTest(new IMyTest2("def")));
}
runTestImplementationMacro(IProtocolTest);
```

RELATED CONCEPTS

Application Testing Overview
"Chapter 3. Test Framework Components" on page 17

RELATED TASKS

"Create a Base Test Class" on page 18
"Set Up the Environment" on page 19

Auxiliary Test Classes

The Test Framework defines two auxiliary classes: ITieredTextBuffer and IArgumentDictionary.

ITieredTextBuffer
ITieredTextBuffer behaves like the C++ ostream class. It contains << operators for all basic types. It can filter output so that detailed information is suppressed or displayed. The text is printed directly to the console (and log file, if the log option is specified). Each instance of ITest contains an ITieredTextBuffer to which subclasses may stream diagnostic text messages. ITest uses this mechanism to report progress and results.

IArgumentDictionary
IArgumentDictionary is a general-purpose class for parsing text arguments on a command line. It takes as input an ordered collection of IString objects and parses them as arguments on a command line into pairs of keys and values. This allows

you to quickly check for the existence of a keyword on the command line or to retrieve the value given for a certain option.

A leading-hyphen character identifies keywords. Anything without a leading hyphen is a value argument. A keyword picks up the following argument if it is not another keyword.

This example shows how IArgumentDictionary parses the command-line input to a test to create the key-value pairs:

```
-parm -n sample1 Joan Tom -ccc 84 85
```

The following table explains the options used in the example.

Key	Value	Description
-parm	*empty IText object*	The value associated with -parm is an empty IText, not NIL. This allows you to distinguish “There is no -parm keyword” from “There is a -parm argument with no associated value.”
-n	sample1	The -n argument picks up the following argument, “sample1,” as its value.
1	Joan	There is no associated keyword, so “Joan” is assigned key 1. The key is an IText object, not a numeric value.
2	Tom	There is no associated keyword, so “Tom” is assigned key 2. The key is an IText object, not a numeric value.
-ccc	84	The -ccc argument picks up the following argument, “84,” as its value.
3	85	There is no associated keyword, so “85” is assigned key 3. The key is an IText object, not a numeric value.

You also may use the specifyNakedOptions function to ensure that certain keywords never take value arguments. Such keywords are called *naked options*. Here is another example:

```
-s 20 -l -o sampout
```

The following table explains the options used in the example.

Key	Value	Description
-s	20	The -s argument picks up the following argument, “20,” as its value. The value must be between 1 and 256.

Key	Value	Description
-l	*no arguments*	The -l represents the login option, which takes no arguments. If a value follows the -l, the value is taken to be a keyless option and is assigned the key 1. The key is an IText object, not a numeric value.
-o	sampout	The -o represents the output-file option, which takes a single argument.

RELATED CONCEPTS

Application Testing Overview
"Chapter 3. Test Framework Components" on page 17

RELATED TASKS

"Print, Display, and Store Test Results"
"Display Internal Test Information" on page 33

Print, Display, and Store Test Results

To write out diagnostic text from a test, use the OutputTextStream function, which recognizes the standard C++ << operator for all built-in types. Text output produced with OutputTextStream displays to the console and is also saved in a ITieredTextBuffer within the test. You can then log the test, including the text buffer. If a test fails, you can retrieve the associated diagnostic text to determine the cause of the failure.

OutputTextStream returns a pointer to a ITieredTextBuffer object that contains the text. Objects of the class ITieredTextBuffer act like C++ ostream objects. They support << operators for all built-in types and for some basic VisualAge C++ types.

Output text is printed via ITieredTextBuffer. You can acquire the ITieredTextBuffer of the test by calling outputTextStream().

```
void IMyTest::test() // This method is in a subclass of ITest
{
    outputTextStream() << "hello, world";
}
```

"Hello, world" is printed out to console when test is running. Information is marked with a level of importance. The text objects in a text buffer are ITieredText objects, so you can show a specified amount of detail when you later retrieve the text. You can also filter out information beyond a specific level of detail.

ITieredTextBuffer allows you to set the tier or indentation for the text. The tier levels are defined as follows in ITieredTextBuffer:

- kTop - Top tier, most general information
- kHeadline - One-line information
- kGeneral - Information of general interest
- kNormal - Default tier

- kDetail - Detailed information not usually needed
- kDebug - Information only needed during debugging
- kBottom - Most detailed information

Most text occupies the tier kNormal. Important information that should not be missed occupies tier kGeneral or kHeadline. Detailed information that can usually be ignored occupies tier kDetail or kDebug.

```
void ISecondTest::test() // This method is in a subclass of ITest{
     ITieredTextBuffer& out = outputTextStream();
     out.pushTier(ITieredTextBuffer::kHeadline);
out << "running isecondtest\n";
out.setrelativeindent(1) << '\n';
out << "options are: [-c <timing count>] [-s <sample count>]\n";
out.setRelativeIndent(-1);
     out.popTier();
}
```

RELATED CONCEPTS

Application Testing Overview
"Chapter 3. Test Framework Components" on page 17
"Auxiliary Test Classes" on page 30

RELATED TASKS

"Display Internal Test Information"

Display Internal Test Information

The Test Framework provides a means by which test classes can display internal information in a textual format. This mechanism works in two ways:

- Callers who want to display tests use the operator<<(ITieredTextBuffer, ITest) function. This is a global function, called whenever a test is run to display the final state of the test. It is the primary interface by which tests present themselves in a human-readable form.
- Writers of ITest subclasses who wish to display data present in their subclass but absent from their base class should override the virtual framework function ITest::print. In their overriding function, they should always call the inherited print function first, so that base-class information is displayed.

```
class IMyTest: public ITest
{
    ...
protected:
    void print(ITieredTextBuffer&);
private:
    double fImportantValue; // This is important and should be shown
};
void IMyTest::print(ITieredTextBuffer& out)
{
    ITest::print(out); // First show base class information
    out << "tmytest important value=" << fImportantValue << '\n';
    // Always end with a '\n'.
}
```

The content of the print function is printed only when you set the echo tier level equal to or below ITieredTextBuffer::kDetail (you set it through the command line). Remember that the default echo tier level is ITieredTextBuffer::kNormal.

```
mytest -t IMyTest -e dTest IMyTest (PASS) // sets echo tier level
{    run count: 1
info: {}
input: {}
TMyTest important value=100
}
```

If you don't include the -e option, you will see this

```
mytest -t IMyTest
Test IMyTest (PASS) {}
```

RELATED CONCEPTS

Application Testing Overview
"Chapter 3. Test Framework Components" on page 17
"Auxiliary Test Classes" on page 30

RELATED TASKS

"Print, Display, and Store Test Results" on page 32

Test Macros

The Test Framework defines macros that you may use to create an application that runs tests.

The runTestImplementationMacro defines an application to run a single test. The beginTestFrameworkMacro, runTestMacro, endTestFrameworkMacro, and runTestResultMacro macros defines an application that runs several tests.

Applications created using these macros recognize the following options:

Option	Description
-e[cho] h|g|n|d|D	Set detail of diagnostic output: headline, general, normal, detail, or Debug.
-n numberOfRuns	Run the test numberOfRuns times; default is to run test once.
-o[ptions]	Pass further arguments (after the -o) to the ITest subclass.
-t[est] class	Run the ITest subclass class.
-log [logfilename]	Write the output into logfilename.

The -test option must be specified when running the application.

runTestImplementationMacro
The runTestImplementationMacro macro expands to define the entire main routine to create and run a single test. It takes the name of the test class for its parameter. The test object is created and the run function is called for the test. The main routine returns with a 0 indicating that the test succeeded; otherwise, 1 is returned.

beginTestFrameworkMacro, runTestMacro, endTestFrameworkMacro, runTestResultMacro
Use beginTestFrameworkMacro, runTestMacro, endTestFrameworkMacro, and runTestResultMacro to create an application that can run various tests instead of just one. The following table summarizes each of these macros:

Macro	Description
beginTestFrameworkMacro	Takes argc and argv for its parameters. This macro expands the code to parse the command-line arguments for options.
runTestMacro	Takes the name of the test class for its parameter. This macro expands the code to create and run the text.
endTestFrameworkMacro	Takes no parameters. This macro expands the code to check if a test was run and prints out an error if no test was run.
runTestResultMacro	Returns a boolean value indicating whether test succeeded.

To use these macros you must write your own main routine:

1. In main routine, use beginTestFrameworkMacro with the argc/argv values.
2. Use runTestMacro for each test class you wish to run.
3. Use the endTestFrameworkMacro and runTestResultMacro to get the result of the test.

Only the test specified with the -t option is run when the application executes.

RELATED CONCEPTS

Application Testing Overview
"Chapter 3. Test Framework Components" on page 17

RELATED TASKS

"Example: Use Test Macros"

Example: Use Test Macros

To create an application that can run various tests instead of just one, use the beginTestFrameworkMacro, runTestMacro, endTestFrameworkMacro, and runTestResultMacro in your own main routine:

```
#include <itest.hpp>
#include <runtest.hpp>
class IMyTest: public ITest
{
public:
    ...
protected:
    ...
};
class ISecondTest: public ITest
{
public:
    ...
protected:
    ...
};
int main(int argc, char **argv)
{
    beginTestFrameworkMacro(argc, argv);
    runTestMacro(IMyTest);
    runTestMacro(ISecondTest);
    endTestFrameworkMacro();
    return (!runTestResultMacro());  // exit with 0 if test succeeded.
}
```

If the code above were compiled and linked to create the application **RunDummyTests**, you would execute the application in the following way to run either `IMyTest` or `ISecondTest`:

```
RunDummyTests -t IMyTest -o "arguments to IMyTest are here"
RunDummyTests -t ISecondTest -n 3 -e d -o "arguments to ISecondTest are here"
```

RELATED CONCEPTS

Application Testing Overview
"Chapter 3. Test Framework Components" on page 17
"Test Macros" on page 34

RELATED TASKS

"Create a Base Test Class" on page 18
"Set Up the Environment" on page 19