

---

# **Migrating from Non-C++ Standard Library Builds to C++ Standard Library Builds**

[www.roguewave.com](http://www.roguewave.com)



# Migrating from Non-C++ Standard Library Builds to C++ Standard Library Builds

---

## Table of contents

<b>INTRODUCTION</b> .....	<b>1</b>
OVERVIEW .....	1
ROGUE WAVE SUPPORT FOR PORTING APPLICATIONS .....	2
<b>HOW TO NAME C STYLE HEADERS</b> .....	<b>2</b>
<b>IOSTREAMS</b> .....	<b>2</b>
<b>SYMBOL REFERENCING AND NAMESPACES</b> .....	<b>2</b>
AVOIDING “ <code>USING NAMESPACE STD</code> ” .....	2
REFERENCING SYMBOLS IN THE <code>STD</code> NAMESPACE.....	3
USING C-STYLE SYMBOLS .....	3
<b>DEPRECATED CONTAINER CLASSES</b> .....	<b>3</b>
CHANGES TO CLASS NAMES .....	4
CHANGES TO THE API .....	4
<i>Using <code>RWDefHArgs(T)</code></i> .....	5
INHERITANCE HIERARCHY CHANGES .....	8
LESS THAN AND EQUALITY SEMANTICS.....	8
<i>Using <code>RWDefCArgs</code></i> .....	9

## Introduction

---

As of Edition 4, SourcePro C++ supports C++ Standard Library builds only. Support for non-standard library builds has been left in the code base, but is deprecated. You are asked to please migrate your code to use standard library builds.

This guide offers information and support for migrating your applications from non-standard library builds to standard library builds.

---

**NOTE:** The terms “standard library” or just “the standard” are used throughout this document to mean the C++ Standard Library.

---

## Overview

When migrating your code to standard library builds, the following issues must be considered. These are discussed in more detail in the next section.

- ◆ [How to Name C Style Headers](#)

With the advent the ISO/ANSI Standard C++ library, the traditional use of the `.h` file extension has been dropped. We recommend that developers adopt the new header file naming convention.

- ◆ [Iostreams](#)

Some Rogue Wave features, such as `iostreams`, have been completely redesigned to conform to the standard and are now incompatible with the traditional implementation.

- ◆ [Symbol Referencing and Namespaces](#)

As the compiler vendors have increasingly conformed to the C++ language standard symbol referencing rules, programmers must take greater care to correctly reference C-style symbols, because the symbols exist in both the global namespace with the older C-style header file naming convention, and in the `std` namespace with the standard library style header file naming convention.

This results in two issues:

- ◆ The C-style headers are prefixed with a `c`. For example, the C-style header `<wchar.h>` becomes the ANSI C++ header `<cwchar>`. If you are using the standard, then C-style symbols are found both in the global namespace and the `std` namespace.
- ◆ When referring to namespace-scope symbols outside the declaring namespace (in the absence of any using declarations or directives), the symbol names must be qualified with the namespace containing the declaration of the symbol.
- ◆ [Deprecated Container Classes](#)

In its support of standard library builds, Rogue Wave has deprecated some of its traditional container classes. However, to ease customers’ transition from non-standard builds to standard builds, some of the supported container classes now contain macro replacements for the deprecated containers, as well as several helper macros.

## Rogue Wave Support for Porting Applications

The Essential Tools Module of SourcePro Core currently offers the capability of using certain classes without any standard library code or dependencies. However, the use of these classes is not supported or guaranteed. If you want help either porting your application to the new SourcePro code using the standard library, or porting the new SourcePro code to a non-standard library build, contact Technical Services.

Please note that the Advanced Tools, XML Streams, Web Services, and Internationalization Modules were all designed to depend on a standard library, so porting these modules to a non-standard build will be more difficult and expensive.

## How to Name C Style Headers

---

The ANSI C++ Standard still supports standard headers with both the traditional `.h` extension or the new naming convention (using the `c` prefix and no `.h` suffix), depending on your compiler.

However, when using C-style headers, because C-style symbols can exist both in the global namespace and the `std` namespace, we recommend that you use the new naming convention for C-style header `include` directives.

## iostreams

---

For `iostreams`, you *must* use the new ISO/IEC 14882:1998 and ANSI Standard `iostream` and the new header file naming. There are no shortcuts or workarounds. The `iostream` library affects all clients migrating code to the new ISO/ANSI C++ Standard to which the Rogue Wave Standard C++ Library Module conforms.

The `.h` is now dropped from the `include` directive as follows:

```
#include <iostream.h> // incorrect
#include <iostream>   // correct
#include <rw/rstream.h> // also correctly includes the iostream
header
```

## Symbol Referencing and Namespaces

---

Because most symbols declared by the standard library are enclosed within the namespace `std`, all references to such symbols outside the namespace (and in the absence of any using declarations or directives) must be qualified with the `std::` prefix. This section recommends methods for meeting this requirement.

### Avoiding “using namespace std”

Unless the scope of a `using` directive is very limited, avoid using the directive in your programs as follows:

```
using namespace std;
```

Doing so injects all symbols declared in the named namespace into the current scope, potentially causing symbol collisions.

Following are some examples of the `using` keyword:

```
using namespace std;    // Avoid. Places the contents of
                        // namespace std into the enclosing
                        // scope.
using ::strlen;        //OK
using std::cout;       //OK
```

## Referencing Symbols in the `std` Namespace

When referencing symbols in the `std` namespace, the safest solution is to scope by individual symbol at the time the symbol is being used. For example:

```
std::vector<int> a;
char buffer[] = "12345";
std::cout << "My test" << std::atoi(buffer) << std::endl;
```

## Using C-Style Symbols

The C-style symbols can exist both in the global namespace and the `std` namespace. Therefore, we recommend that you reference the C-style symbols to the appropriate namespace.

For instance, in the code in the above section, `int std::atoi(const char* )` exists in both `<stdlib.h>` and `<cstdlib>`, and is found both in the global namespace and `std` namespace. So you would use

```
#include <cstdlib>
```

instead of `#include <stdlib.h>` and then scope the function correctly, as

```
int std::atoi(const char* p)
```

## Deprecated Container Classes

---

The standard library container classes differ from the deprecated container classes in four notable ways:

- ◆ [Changes to Class Names](#)
- ◆ [Changes to the API](#)
- ◆ [Inheritance Hierarchy Changes](#)
- ◆ [Less Than and Equality Semantics](#)

This section lists the deprecated classes and their equivalent supported classes, and includes options for migrating to the new classes.

### Deprecated container classes and their equivalent standard classes

Deprecated Class	Equivalent Class Supported by Standard C++ Library
RWTPtrDlist<T>	RWTPtrDlist<T,A>
RWTPtrDlistIterator<T>	RWTPtrDlistIterator<T,A>
RWTPtrHashSet<T>	RWTPtrHashSet<T,H,EQ,A>
RWTPtrHashSetIterator<T>	RWTPtrHashSetIterator<T,H,EQ,A>
RWTPtrHashDictionary<K,V>	RWTPtrHashMap<K,T,H,EQ>
RWTPtrHashDictionaryIterator<K,V>	RWTPtrHashMapIterator<K,T,H,EQ>
RWTPtrOrderedVector<T>	RWTPtrOrderedVector<T,A>
RWTPtrSlist<T>	RWTPtrSlist<T,A>
RWTPtrSlistIterator<T>	RWTPtrSlistIterator<T,A>
RWTPtrSortedVector<T>	RWTPtrSortedVector<T,C,A>
RWTValDlist<T>	RWTValDlist<T,A>
RWTValDlistIterator<T>	RWTValDlistIterator<T,A>
RWTValHashSet<T>	RWTValHashSet<T,H,EQ,A>
RWTValHashSetIterator<T>	RWTValHashSetIterator<T,H,EQ,A>
RWTValHashDictionary<K,V>	RWTValHashMap<K,T,H,EQ>
RWTValHashDictionaryIterator<K,V>	RWTValHashMapIterator<K,T,H,EQ>
RWTValOrderedVector<T>	RWTValOrderedVector<T,A>
RWTValSlist<T>	RWTValSlist<T,A>
RWTValSlistIterator<T>	RWTValSlistIterator<T,A>
RWTValSortedVector<T>	RWTValSortedVector<T,C,A>

## Changes to Class Names

You do not need to change the names of your classes. Macro replacements are used internally for all deprecated classes, so you may leave these names in your code as they are.

## Changes to the API

The primary changes to the API in the new container classes are differences in the template parameters for the templated hash containers and the templated sorted containers. These containers now use function objects for hashing and equality.

Some collections take a different number of template parameters, depending on which underlying implementation they use. For example, when *RWTPtrHashSet* is used with the standard library, it takes three template arguments. However, when that same class is used without the standard, the interface calls for only one template parameter, namely the type of item being contained.

To help you write portable code that works with or without the standard, the Essential Tools Module provides two macros, **RWDefHArgs(T)** and **RWDefCArgs(T)**.

- ◆ **RWDefHArgs(T)**

Use **RWDefHArgs(T)**, an abbreviation for Rogue Wave Default Hash Arguments, for the hash-based template collections.

- ◆ **RWDefCArgs(T)**

Use **RWDefCArgs(T)**, an abbreviation for Rogue Wave Default Comparison Arguments, with *RWTPtrSortedVector* and *RWTValSortedVector*.

To migrate the deprecated container classes to supported standard library classes, two options exist:

- ◆ Use the macros **RWDefHArgs** and **RWDefCArgs**, along with the macro replacements for the deprecated container classes.
- ◆ Modify your code to directly use the supported container classes.

Examples illustrating these options are offered below.

## Using RWDefHArgs(T)

Helper macro **RWDefHArgs** and template class *RWTHasher* are provided for the hash containers. The examples in this section illustrate their use.

- ◆ Example 1: Building a container object and using *RWTHasher* as a hashing object
- ◆ Example 2: Building a container object and using *RWTHasher* as a hashing object, with the hashing instance constructed as a temporary object
- ◆ Example 3: Using `RWCString::hash` function directly
- ◆ Example 4: Using `RWCString::hash` function directly in a template-like manner

### Notes on the Examples

- ◆ For `<int RWDefHArgs(int)>`, the **RWDefHArgs** expands to `<int ,RWTHasher<int>, equal_to<int> >`
- ◆ *RWTHasher* defines an interface only, that is, a function pointer. The programmer must provide the hash function, usually to the constructor. For more details on the *RWTHasher*, please see the implementation in `rw/edefs.h`
- ◆ The `equal_to` used here is the standard library `std::equal_to` which calls the equality operator `== ()` for the class.

*Example 1: Build a container object and use RWTHasher as a hashing object*

```
#include <rw/tphasht.h>
#include <rw/cstring.h>
#include <iostream>

// Custom hash function - does nothing, returns 0
unsigned custom_hash (const RWCString&)
{
    return 0;
}

int main()
{
    //Construct an RWTHasher<RWCString> object and pass the address of
    //the custom_hash function, and use it in constructing the
    //container.
    RWTHasher<RWCString> h (&custom_hash);

    // Construct the container and use a constructor that allows us to
    //specify what "hashing" object we want the container to use
    RWTPtrHashMultiSet<RWCString,
                      RWTHasher<RWCString>,
                      std::equal_to<RWCString> > s0 ( h );

    // Same with using the RWDefHArgs macro
    RWTPtrHashMultiSet<RWCString RWDefHArgs(RWCString) > s1 ( h );

    s0.insert(new RWCString("foo"));
    s1.insert(new RWCString("bar"));
    return 0;
}
```

*Example 2: Build a container object and use RWTHasher as a temporary hashing object*

```
// The hashing object is constructed as a temporary
// Custom hash function - does nothing, returns 0
#include <rw/tphasht.h>
#include <rw/cstring.h>

unsigned custom_hash (const RWCString&)
{
    return 0;
}

int main()
{
    typedef RWTPtrHashMultiSet<RWCString,
                              RWTHasher<RWCString>,
                              std::equal_to<RWCString> > MY_Map;

    // Same as above but using the convenience macro RWDefHArgs
    typedef RWTPtrHashMultiSet<RWCString RWDefHArgs(RWCString) >
        MY_Map_Macro;

    MY_Map s1 = MY_Map(RWTHasher<RWCString> (&custom_hash));
    MY_Map_Macro s3 = MY_Map_Macro( RWTHasher<RWCString>
                                   (&custom_hash));

    s1.insert(new RWCString("bar"));
    s3.insert(new RWCString("bar"));
    return 0;
}
```

### *Example 3: Use RWCString::hash function directly*

```
unsigned custom_hash (const RWCString& _ref)
{
    return _ref.hash ();
}

int main()
{
    RWTHasher<RWCString> h (&custom_hash);
    RWTPtrHashMultiSet<RWCString,
                      RWTHasher<RWCString>,
                      std::equal_to<RWCString> > s0 ( h );

    // Same as above with macro RWDefHArgs
    RWTPtrHashMultiSet<RWCString RWDefHArgs(RWCString) > s1 ( h );

    s0.insert(new RWCString("foo"));
    s1.insert(new RWCString("bar"));
    return 0;
}
```

### *Example 4: Use RWCString::hash function directly in a template-like manner*

```
template<class T>
unsigned t_hash (const T& t)
{
    return t.hash ();
}

// Explicit instantiation of t_hash for type RWCString
template unsigned t_hash<RWCString> (const RWCString&);

int main()
{
    RWTHasher<RWCString> h (&t_hash<RWCString>);
    RWTPtrHashMultiSet<RWCString,
                      RWTHasher<RWCString>,
                      std::equal_to<RWCString> > s0 ( h );

    // Same as above, using the convenience macro RWDefHArgs
    RWTPtrHashMultiSet<RWCString RWDefHArgs(RWCString) > s1 ( h );

    s0.insert(new RWCString("foo"));
    s1.insert(new RWCString("foo"));
    return 0;
}
```

## Inheritance Hierarchy Changes

The standard library-based container classes have a different hierarchy than the deprecated classes.

If you have existing code that makes use of any of the inheritance relationships among the collection class templates, that code will no longer compile. The inheritance relationships among the deprecated container classes are unique and have no counterpart in the standard library-based implementations.

Significant code changes may need to be made for:

RWTPtrHashSet

RWTVaHashSet

RWTPtrHashTable

RWTVaHashTable

RWTPtrOrderedVector

RWTVaOrderedVector

RWTPtrSortedVector

RWTVaSortedVector

For the change in hierarchy, there are no easy solutions. You must make code changes, as the underlying functionality no longer exists within the inheritance hierarchy.

## Less Than and Equality Semantics

The less than and equality semantics requirements have not changed, but the less-than must be declared for use with the underlying standard container. The helper macro **RWDefCArgs** in the examples in this section demonstrates how the less-than operator must be used.

The operator `==` must be defined for classes used in the containers below.

RWTPtrDlist

RWTVaDlist

RWTPtrOrderedVector

RWTVaOrderedVector

RWTPtrSlist

RWTVaSlist

The operator `<` and `==` must be defined for classes used in the containers below.

RWTPtrSortedVector

RWTVaSortedVector

The following two examples demonstrate the use of **RWDefCArgs** and the equality operator `==` for user-defined classes.

## Using RWDefCArgs

The examples in this section demonstrate the use of **RWDefCArgs** with classes contained in the **RWPtrSortedVector** container. This macro is used with **RWPtrSortedVector** and **RWValSortedVector**.

This section includes an example that defines the equality operator `==()` and the less-than operator `<()`.

### Notes on the Examples

- ◆ `<T RWDefCArgs(T)>` expands to `<T, RW_SL_STD(less)< T >>`
- ◆ “less” is the standard library `std::less` functor. This functor calls the less-than operator `<()` for the class.
- ◆ Not defining the operator `<` will result in a compilation error for user-defined classes or structs. This is a requirement of the `std::less` functor and the **RWPtrSortedVector** and **RWValSortedVector** container classes.

### Example 1: RWDefCArgs

```
// RWDefCArgs Example
#include <rw/tpsrtvec.h>
#include <rw/cstring.h>
#include <iostream>

// User defined comparison operator
// A struct with the comparison operator performing a string
comparison
struct user_defined
{
    user_defined () : i_ (0) {
    }

    user_defined (int i) : i_ (i) {
    }

    bool operator< (const user_defined& right) const
    {
        return i_ < right.i_;
    }

    int    i_;
};

// User defined hash function
unsigned ud_custom_hash (const user_defined& ref)
{
    return ref.i_;
}

// Custom hash function - does nothing
// This uses the RWCString hash, but could use any hash function
unsigned string_custom_hash (const RWCString& ref)
{
    return ref.hash ();
}

int main()
{
    // Test "user_defined" comparison operator
    // Dependent on the class
}
```

```

    {
        user_defined a (1), b (2);
        std::cout << (a < b) << std::endl;

        user_defined c (1), d (1);
        std::cout << (c < d) << std::endl;
    }

// Test "user_defined" comparison operator with the container class
{
    RWTPtrSortedVector<user_defined RWDefCArgs(user_defined) > s;

    user_defined* str0 = new user_defined (1);
    user_defined* str1 = new user_defined (2);
    user_defined* str2 = new user_defined (3);

    s.insert(str0);
    s.insert(str1);
    s.insert(str2);

    RWTPtrSortedVector<user_defined RWDefCArgs(user_defined) >
::iterator it;
    for (it = s.begin (); it != s.end (); it++)
        std::cout << (*it)->i_ << std::endl;
}

// Test RWCString comparison operator with the container class
{
    RWTPtrSortedVector<RWCString RWDefCArgs(RWCString) > s;

    RWCString* str0 = new RWCString ("foo");
    RWCString* str1 = new RWCString ("bar");
    RWCString* str2 = new RWCString ("foobar");

    s.insert(str0);
    s.insert(str1);
    s.insert(str2);

    RWTPtrSortedVector<RWCString RWDefCArgs(RWCString) >
::iterator it;
    for (it = s.begin (); it != s.end (); it++)
        std::cout << (*it)->data () << std::endl;
}

    return 0;
}

```

### Example 2: Equality operator == ()

This example demonstrates the equality operator == () for user-defined classes contained in the **RWTPtrHashMultiSet** container. Notes on the Example

- ◆ <T RWDefHArgs(T)> expands to <T, RWTHasher<T>, equal\_to<T>>
- ◆ equal\_to is the C++ Standard Library std::equal\_to functor. This functor calls the operator == () for the class object.
- ◆ Not defining the operator == will result in a compilation error for user-defined classes or structs. This is a requirement of the std::equal\_to functor.

```

#include <rw/tphasht.h>
#include <rw/cstring.h>
#include <iostream>

#include <cstring> // for strcmp

// RWCString example
// A struct with the equality operator performing a string comparison

```

```

struct need_eq
{
    need_eq () : s_ (NULL) {
    }

    need_eq (const char* s) : s_ (s) {
    }

    const char* s_;

    bool operator == (const need_eq& right) const {
        return 0 == std::strcmp (s_, right.s_);
    }
};

// A struct whose equality operator does a plain comparison between
// members
struct dont_need_eq
{
    dont_need_eq () : i_ (0), d_ (0.0) {
    }

    dont_need_eq (int i, double d) : i_ (i), d_ (d) {
    }

    bool operator== (const dont_need_eq& right) const {
        return i_ == right.i_ && d_ == right.d_;
    }

    int    i_;
    double d_;
};

// hash for the dont_need_eq_custom class
// Custom hash function
unsigned dont_need_eq_custom_hash (const dont_need_eq& ref)
{
    return ( ref.i_ );
}

// hash for the need_eq class
// Custom hash function
unsigned need_eq_custom_hash (const need_eq& ref)
{
    return (ref.s_ [0] * 27) + ref.s_ [1];
}

// RWCString hash used
// Custom hash function
unsigned string_custom_hash (const RWCString& ref)
{
    return ref.hash ();
}

// convenience macro used in container declaration. See below
// This is very similar to the way Rogue Wave uses this
// the macro RWDefHArgs(T)
#define HashEqual(T) RWHasher< T > , custom_equal

struct custom_equal
{
    bool operator () (const RWCString& left, const RWCString& right)
const {
        return left == right;
    }
};

```

```

int main()
{
    // Test the equality operator for the class "dont_need_eq"
    {
        dont_need_eq a (1, 1.0), b (2, 2.0);
        std::cout << (a == b) << std::endl;

        dont_need_eq c (1, 1.0), d (1, 1.0);
        std::cout << (c == d) << std::endl;
    }

    {
        // Construct an RWTHasher<dont_need_eq> object and pass the
        // address of the dont_need_eq_custom_hash function to it
        RWTHasher<dont_need_eq> h (&dont_need_eq_custom_hash);

        // Construct the container; use our custom_equal functor
        RWTPtrHashMultiSet<dont_need_eq RWDefHArgs(dont_need_eq) >
s ( h );

        dont_need_eq* s0 = new dont_need_eq (1,1.2);
        dont_need_eq* s1 = new dont_need_eq (2,3.3);
        dont_need_eq* s2 = new dont_need_eq (3,4.4);

        s.insert(s0);
        s.insert(s1);
        s.insert(s2);

        std::cout << "Search (1,1.2)      : "
        << (s.find (s0)?true:false) << std::endl;

        std::cout << "Search (2,3.3)      : "
        << (s.find (s1)?true:false) << std::endl;

        std::cout << "Search (3,4.4)      : "
        << (s.find (s2)?true:false) << std::endl;
    }

    const char* one = "foo";
    const char* two = "bar";

    // Test the equality operator for the class "need_eq"
    {
        need_eq a (one), b (two);
        std::cout << (a == b) << std::endl;

        need_eq c (one), d (one);
        std::cout << (c == d) << std::endl;
    }

    {
        // Construct an RWTHasher<RWCString> object and pass the
        // address of the string_custom_hash function to it
        RWTHasher<RWCString> h (&string_custom_hash);

        // Construct the container and use our custom_equal functor
        RWTPtrHashMultiSet<RWCString, HashEqual(RWCString) > s ( h );

        RWCString* str0 = new RWCString ("foo");
        RWCString* str1 = new RWCString ("bar");
        RWCString* str2 = new RWCString ("foobar");

        s.insert(str0);
        s.insert(str1);
        s.insert(str2);

        std::cout << "Search \"foo\"      : "
        << (s.find (str0)?true:false) << std::endl;
    }
}

```

```

        std::cout << "Search \"bar\"      : "
                  << (s.find (str1)?true:false) << std::endl;

        std::cout << "Search \"foobar\" : "
                  << (s.find (str2)?true:false) << std::endl;
    }

    {
        // Construct an RWTHasher<RWCString> object and pass the
// address of
        // the string_custom_hash function to it
        RWTHasher<RWCString> h (&string_custom_hash);

        // Construct the container; use stdlib equal_to
        RWTPtrHashMultiSet<RWCString RWDefHArgs(RWCString) > s ( h );

        RWCString* str0 = new RWCString ("foo");
        RWCString* str1 = new RWCString ("bar");
        RWCString* str2 = new RWCString ("foobar");

        s.insert(str0);
        s.insert(str1);
        s.insert(str2);

        std::cout << "Search \"foo\"      : "
                  << (s.find (str0)?true:false) << std::endl;

        std::cout << "Search \"bar\"      : "
                  << (s.find (str1)?true:false) << std::endl;

        std::cout << "Search \"foobar\" : "
                  << (s.find (str2)?true:false) << std::endl;
    }

    {
        // Construct an RWTHasher<RWCString> object and pass the
// address of the need_eq_custom_hash function to it
        RWTHasher<need_eq> h (&need_eq_custom_hash);

        // Construct the container; use stdlib equal_to
        RWTPtrHashMultiSet<need_eq RWDefHArgs(need_eq) > s ( h );

        need_eq* str0 = new need_eq ("foo");
        need_eq* str1 = new need_eq ("bar");
        need_eq* str2 = new need_eq ("foobar");

        s.insert(str0);
        s.insert(str1);
        s.insert(str2);

        std::cout << "Search \"foo\"      : "
                  << (s.find (str0)?true:false) << std::endl;

        std::cout << "Search \"bar\"      : "
                  << (s.find (str1)?true:false) << std::endl;

        std::cout << "Search \"foobar\" : "
                  << (s.find (str2)?true:false) << std::endl;
    }

    return 0;
}

```

© Copyright Rogue Wave Software, Inc. 2002. All Rights Reserved.  
Rogue Wave and .h++ are registered trademarks of Rogue Wave Software, Inc.  
All other trademarks are the property of their respective owners.



**Corporate Headquarters**  
Toll-free: (800) 487-3217  
E-mail: [sales@roguewave.com](mailto:sales@roguewave.com)

[www.roguewave.com](http://www.roguewave.com)

**The Netherlands**

Rogue Wave Software B.V.  
Telephone: +31 23 542 2873

**Germany**

Rogue Wave Software GmbH  
Telephone: +49 6103-59 34-0

**France**

Rogue Wave Software S.A.R.L.  
Telephone: +33 1 41 96 26 26

**United Kingdom**

Rogue Wave Software U.K. Ltd.  
Telephone: +44 118 9358600

**Italy**

Rogue Wave Software S.R.L.  
Telephone: +39 02 4125.081

**Japan**

Rogue Wave Software Japan K.K.  
Telephone: +81 3 3512-5012  
E-mail: [jpinfo@roguewave.com](mailto:jpinfo@roguewave.com)  
[www.roguewave.co.jp](http://www.roguewave.co.jp)

