

NewJ Library for C++ Developer's Guide



PureNative Software

www.pure-native.com

Copyright © 2002-2004 PureNative Software Corporation. All rights reserved.

PureNative, the PureNative logo, NewJ, the NewJ logo, NewJNI, NewJNIInterop, and Pie are trademarks of PureNative Software Corporation. Java is a registered trademark of Sun Microsystems, Inc. All other trademarks and registered trademarks are property of their respective owners.

The software described herein is licensed. Unauthorized reproduction or distribution of this software, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under law. Consult accompanying license agreement for additional permissions and restrictions.

NewJ Library for C++ is based on the Java 2 Platform, Standard Edition, version 1.2.2 API Specification from Sun Microsystems, Inc. *NewJNI* and *NewJNIInterop* are compatible with the JNI Specification from Sun Microsystems, Inc., and support any JNI-compatible version of the Java platform, including version 1.2, 1.3, or later. *NewJ Library for C++* is not a product of nor endorsed by Sun Microsystems, Inc.

Patent Pending.

Developer release 2: May 2004.

{7} Using NewJ Library Along With Other C/C++ Libraries

Integrating NewJ Library With Other C/C++ Libraries

Although simple console and non-interactive applications may be developed entirely using NewJ Library, for more complex visual or platform-rich applications you may wish to use other C/C++ libraries in addition to NewJ Library. On the Windows platform, these libraries may include the Win32 API or MFC. Or, you may be maintaining an existing legacy application and wish to take advantage of NewJ Library in new code that you write. For general purpose, you may wish to leverage the ANSI C++ Standard Library, including STL, in order to benefit from all that standard C++ has to offer in combination with the Java API and language features provided by NewJ Library. This chapter explains how to do all these things—to use NewJ Library along with other C and C++ libraries in your new or existing applications. Lastly, this chapter will give examples of using NewJ Library along with the following C/C++ libraries and APIs: STL, Win32 API, MFC.

PieSystem: The Class That Bridges the Java API and Language with C++

According to the vaunted principles of good object-oriented design, the Java language and API hide access to low-level constructs, such as pointers to null-terminated strings, pointers to array storage, and native operating system handles. But these are precisely the things that are needed by most, if not all, C and C++ libraries and APIs. How then can you gain access to them? Traditional Java virtual machine environments provide access to these constructs via Java Native Interface (JNI). However, JNI programming can be cumbersome and error-prone, generally favors C over C++, and incurs measurable overhead in order to call between native and VM environments. Due to these factors and the unique requirements of a 100% native environment, NewJ Library provides the PieSystem class instead. It bridges the Java API and language with ANSI C++ in a way that is natural to both C++ and Java developers. Above all, it is ultra-efficient and does not incur any cross-environment overhead.

The PieSystem class provides a variety of functionality for native developers. This functionality includes:

- initialization and termination of the Pie Run-time Library
- creating new String objects using low-level pointers
- accessing low-level pointers to string and array storage
- single-byte/multi-byte/Unicode string conversion
- accessing low-level operating system-specific handles

We will now cover each of these areas of functionality in turn.

Initialization and Termination of Pie Run-time Library

New applications developed with NewJ Library generally use one of the starters detailed in chapter 4, “Pie Run-time Library Reference.” These starters take care of initialization and termination for you without any intervention on your part. However, you may wish to manually initialize or terminate the Pie Run-time Library. This may be required for integrating NewJ Library with new or legacy applications, which make use of other C/C++ libraries along with NewJ.

To initialize the Pie Run-time Library, use the `initialize()` method as follows:

```
if (! pie::PieSystem::initialize(0, 0))
    {
    // Failed to initialize Pie Run-time Library
    exit(-1);
    }
```

Note that the two parameters for `initialize()` are reserved and should both be 0. The `initialize()` method returns true indicating that initialization has succeeded or false if it fails. You should always check for failure when calling `initialize()`.

To terminate the Pie Run-time Library, use the `terminate()` method as follows:

```
pie::PieSystem::terminate();
```

Creating New String Objects Using Low-level Pointers

`PieSystem` supports various methods for creating `String` objects from low-level pointers to strings and character arrays. These methods allow you to create `String` objects from the following:

- single-byte/multi-byte null-terminated character strings
- single-byte/multi-byte character arrays, which need not be null-terminated
- Unicode null-terminated character strings
- Unicode character arrays, which need not be null-terminated

The single-byte/multi-byte character strings and character arrays are encoded using the platform's default character encoding. The Unicode character strings and character arrays use the UTF-16 format.

You can create a `String` object from a null-terminated byte string as follows:

```
StringR aString = pie::PieSystem::toString(myByteString);
```

where *myByteString* is a pointer to the null-terminated byte string (`const pi_byte*`).

Create a `String` object from a byte array, which may or may not be null-terminated, like so:

```
StringR aString = pie::PieSystem::toString(myByteArray, myByteArrayLength);
```

where *myByteArray* is a pointer to the byte array (const pi_byte*), and *myByteArrayLength* is the length (pi_int) of the byte array.

You can create a String object from a null-terminated Unicode string as follows:

```
StringR aString = pie::PieSystem::toString(unicodeString);
```

where *unicodeString* is a pointer to the null-terminated Unicode string (const pi_char*). Keep in mind that if you wish to create a String object from a string literal you must use the PI_T() macro instead of toString(); this guarantees that the string literal is treated efficiently and properly.

To create a String object from a Unicode character array, use the following form:

```
StringR aString = pie::PieSystem::toString(unicodeArray, unicodeArrayLength);
```

where *unicodeArray* is a pointer to the Unicode character array (const pi_char*), which may or may not be null-terminated, and *unicodeArrayLength* is the length (pi_int) in characters (not bytes) of the array.

Accessing Low-level Pointers to String and Array Storage

PieSystem provides several methods to access low-level pointers to String object and array storage. These include the methods getCString(), getByteArrayElements(), and getCharArrayElements(), to name a few.

The getCString() method provides read-only access to the null-terminated Unicode character string for a specified String object. It is similar to basic_string::c_str() in the C++ Standard Template Library (STL). getCString() is declared as follows:

```
static const pi_char* getCString(::java::lang::String* string);
```

where *string* is the String object to be converted. Note that the lifetime of the String object must be maintained externally and that the returned pointer will be valid as long as the String object stays in existence. Further information may be found in the documentation comments for PieSystem.

Here is an example of using getCString():

```
StringR simpleString = PI_T("Simple string");  
const pi_char* simpleStringPtr = ::pie::PieSystem::getCString(*simpleString);  
int count = wcslen(simpleStringPtr);
```

To access low-level pointers to array storage, the following methods are declared:

```
static pi_byte* getByteArrayElements(const PieArrayBase* arrayBase);
```

```

static pi_char* getCharArrayElements(const PieArrayBase* arrayBase);

static pi_short* getShortArrayElements(const PieArrayBase* arrayBase);

static pi_int* getIntArrayElements(const PieArrayBase* arrayBase);

static pi_long* getLongArrayElements(const PieArrayBase* arrayBase);

static pi_float* getFloatArrayElements(const PieArrayBase* arrayBase);

static pi_double* getDoubleArrayElements(const PieArrayBase* arrayBase);

static bool* getBooleanArrayElements(const PieArrayBase* arrayBase);

static ::java::lang::ObjectR* getObjectArrayElements(const PieArrayBase* arrayBase);

static PieInterfaceReference< void >* getInterfaceArrayElements(const
PieArrayBase* arrayBase);

```

where *arrayBase* is a pointer to an array, which is in actuality an Object-derived type. These methods return pointers to writable (non-const) storage. However, it is recommended that you generally treat these pointers as read-only (const) storage just like `getCString()`.

Here is an example of using one of the array access methods, in this case, `getByteArrayElements()`:

```

StringR simpleString = PI_T("Simple string");
pi_byteAR byteArray = simpleString->getBytes();
pi_byte* byteArrayPtr = ::pie::PieSystem::getByteArrayElements(*byteArray);
...

```

Note that the low-level storage pointers returned by `get..ArrayElements()` methods are not necessarily null-terminated. They are only null-terminated if the array itself is null-terminated. Only certain `PieSystem` methods, such as `toByteString()` methods, make null-terminated arrays.

Converting Between Single-byte/Multi-byte Strings and Unicode Strings

`PieSystem` provides several methods to convert between byte strings and Unicode String objects. If you need to convert a byte string to a Unicode String object, simply use one of the `toString()` methods detailed above. On the other hand, the `PieSystem` class provides two specialized methods for converting Unicode String objects to null-terminated byte strings. They are declared as:

```

static pi_byteAR toByteString(::java::lang::StringR text);

```

```
static pi_byteAR toByteString(::java::lang::StringR text, pi_int length);
```

where *text* is the String to be converted and *length* is number of characters to convert.

Here is an example of using `toByteString()`:

```
StringR myString = PI_T("Simple string");
pi_byteAR byteArray = pie::PieSystem::toByteString(myString);
const pi_byte* byteArrayPtr = pie::PieSystem::getByteArrayElements(*byteArray);
int count = strlen( byteArrayPtr );
```

If you need to convert a Unicode String object to a byte string, but you do not require the resultant byte array to be null-terminated, simply use the `getBytes()` method of String. Here is an example of using `getBytes()`:

```
StringR myString = PI_T("Simple string");
pi_byteAR byteArray = myString.getBytes();
const pi_byte* byteArrayPtr = pie::PieSystem::getByteArrayElements(*byteArray);
...
```

Accessing Low-level Operating System-specific Handles

PieSystem provides access to certain low-level operating system-specific handles. These methods are detailed in documentation comments for PieSystem. Likely, the depth and breadth of these methods will grow in the future, especially as the need arises.

Example: Using NewJ Library Along With STL

Earlier examples have shown how to use NewJ Library along with the ANSI C++ Standard Library functions, such as `strlen()`. But what about STL? you may be wondering. The PieSystem class makes it easy to convert NewJ Library types to/from STL types. These conversions include:

- converting a String object to an STL `basic_string`
- converting an STL `basic_string` to a String object
- converting a Pie array to an STL vector
- converting an STL vector to a Pie array

Here is how to convert a String object to an STL string (`basic_string< char >`):

```
StringR simpleString = PI_T("Simple string");

// First, convert String object to null-terminated byte array
pi_byteAR byteString = ::pie::PieSystem::toByteString(simpleString);
```

```
// Obtain pointer to array storage
pi_byte* byteStringPtr = ::pie::PieSystem::getByteArrayElements(*byteString);

// Make a basic_string< char > using the pointer to array storage
std::basic_string< char > basicString(byteStringPtr);
```

Here is how to convert a String object to an STL wstring (basic_string< wchar_t >):

```
// This example assumes that sizeof(wchar_t) == sizeof(pi_char)

StringR simpleString = PI_T("Simple string");

// Obtain pointer to character string storage, similar to basic_string::c_str()
const pi_char* charStringPtr = ::pie::PieSystem::getCString(*simpleString);

// Make a basic_string< wchar_t > using the pointer to array storage
std::basic_string< wchar_t > wideString(charStringPtr);
```

How about converting an STL string to a String object? Here's how:

```
std::string basicString("Sample string");

StringR sampleString = ::pie::PieSystem::toString(basicString.c_str());
```

That's it. Almost the same logic can be used to convert an STL wstring to a String object.

```
// This example assumes that sizeof(wchar_t) == sizeof(pi_char)

std::wstring wideString(L"Sample string");

StringR sampleString = ::pie::PieSystem::toString(wideString.c_str());
```

That covers strings. Now let's move on to converting between Pie arrays and STL vector. Here is an example for converting a Pie array to an STL vector:

```
StringR simpleString = PI_T("Simple string");

pi_byteAR byteArray = simpleString->getBytes();

const pi_byte* bytesPtr = ::pie::PieSystem::getByteArrayElements(*byteArray);

std::vector< char > charVector(bytesPtr, &bytesPtr[byteArray->length_]);
```

How about the other way around? Here is how to convert an STL vector to a Pie array:

```

// This example assumes that sizeof(int) == sizeof(pi_int)

// Make 5 element vector, initializing each element to 1000
std::vector< int > intVector(5, 1000);

pi_intAR intArray = pi_newarray(pi_int, 5);

pi_byte* intsPtr = ::pie::PieSystem::getIntArrayElements(*intArray);

memcpy(intsPtr, &intVector[0], intsPtr->length_ * sizeof(pi_int));

```

Example: Using NewJ Library Along With the Win32 API

You can build on the previous examples for the ANSI C++ Standard Library and STL in order to use NewJ Library along with the Win32 API. For example, many Win32 API functions take null-terminated byte strings as parameters. Here's how to convert a String object to a null-terminated byte string and pass it as a parameter to a Win32 API function:

```

// This example assumes that TCHAR is char (_UNICODE is undefined)

StringR simpleString = PI_T("Hello, Win32");

// First, convert String object to null-terminated byte array
pi_byteAR byteString = ::pie::PieSystem::toByteString(simpleString);

// Obtain pointer to array storage
const pi_byte* byteStringPtr = ::pie::PieSystem::getByteArrayElements(*byteString);

// Pass pointer to array storage as parameter to Win32 API function
::SetWindowText(hWnd, bytesStringPtr);

```

Example: Using NewJ Library Along With MFC

Using NewJ Library along with MFC is very similar to using it with the Win32 API. Here is the Win32 example adapted for MFC:

```

// This example assumes that TCHAR is char (_UNICODE is undefined)

StringR simpleString = PI_T("Hello, MFC");

// First, convert String object to null-terminated byte array
pi_byteAR byteString = ::pie::PieSystem::toByteString(simpleString);

// Obtain pointer to array storage
const pi_byte* byteStringPtr = ::pie::PieSystem::getByteArrayElements(*byteString);

```

```
// Pass pointer to array storage as parameter to Win32 API function
pWnd->SetWindowText(bytesStringPtr);
```

One of the most important conversions for NewJ/MFC developers is between NewJ String objects and MFC String (CString) objects. Here's how to convert a NewJ String to an MFC String:

```
// This example assumes that TCHAR is char (_UNICODE is undefined)

StringR simpleString = PI_T("From NewJ to MFC");

// First, convert String object to null-terminated byte array
pi_byteAR byteString = ::pie::PieSystem::toByteString(simpleString);

// Obtain pointer to array storage
const pi_byte* byteStringPtr = ::pie::PieSystem::getByteArrayElements(*byteString);

// Make a MFC String object
CString mfcString(bytesStringPtr);
```

And here's how to convert an MFC String object to a NewJ String object:

```
// This example assumes that TCHAR is char (_UNICODE is undefined)

CString mfcString(_T("From MFC to NewJ"));

StringR theString = ::pie::PieSystem::toString((LPCTSTR)mfcString);
```