JNI – C++ integration made easy

Evgeniy Gabrilovich gabr@acm.org
Lev Finkelstein lev@zapper.com

Abstract

The Java Native Interface (JNI) [1] provides interoperation between Java code running on a Java Virtual Machine and code written in other programming languages (e.g., C++ or assembly). The JNI is useful when existing libraries need to be integrated into Java code, or when portions of the code are implemented in other languages for improved performance. The Java Native Interface is extremely flexible, allowing Java methods to invoke native methods and vice versa, as well as allowing native functions to manipulate Java objects. However, this flexibility comes at the expense of extra effort for the native language programmer, who has to explicitly specify how to connect to various Java objects (and later to disconnect from them, to avoid resource leak). We suggest a template-based framework that relieves the C++ programmer from most of this burden. In particular, the proposed technique provides automatic selection of the right functions to access Java objects based on their types, automatic release of previously acquired resources when they are no longer necessary, and overall simpler interface through grouping of auxiliary functions.

Introduction

The Java Native Interface[1] is a powerful framework for seamless integration between Java and other programming languages (called "native languages" in the JNI terminology). A common case of using the JNI is when a system architect wants to benefit from both worlds, implementing communication protocols in Java and computationally expensive algorithmic parts in C++ (the latter are usually compiled into a dynamic library, which is then invoked from the Java code). The JNI renders native applications with much of the functionality of Java, allowing them to call Java methods, access and modify Java variables, manipulate Java exceptions, ensure thread-safety through Java thread synchronization mechanisms, and ultimately to directly invoke the Java Virtual Machine.

This functional wealth is provided through a rather complex interface between the native code and the Java programming environment. For example, accessing an integer Java member variable from the native code is a multi-step process, which involves first querying the class type whose variable is to be retrieved (using the JNI function GetObjectClass), obtaining the field identifier (using GetFieldID), and finally retrieving the field per se (using GetIntField). The latter function is a representative of a set of functions (Get<type>Field), each corresponding to a different variable type; thus, accessing a variable requires explicit specification of the appropriate function. To streamline these steps, we suggest using a template (parameterized with a variable type), which encapsulates all these low-level operations, giving the programmer easy-to-use assignment/modification functionality similar to that of native C++ variables. As far as

[1] See the sidebar.

data types are concerned, we develop a complete set of templates for primitive types (jint, jfloat etc.), as well as outline a generic approach for accessing any user-defined type.

Observe also that there are a number of JNI API functions, which are complementary in their nature. For example, if a native function has a parameter of type jstring, it should first convert this parameter into a conventional C++ string of chars using GetStringChars (or GetStringUTFChars), and subsequently explicitly release this temporary representation using the corresponding function ReleaseStringChars (or ReleaseStringUTFChars). This mode of operation could be greatly simplified by implementing a proxy-style smart pointer, which realizes the "construction as resource acquisition" idiom [2]. This way, a Java string is transparently converted on access into a C++ string (char*), and when it later goes out of scope, the smart pointer destructor releases any temporary resources used. The proxy also provides proper handling of the transfer of ownership, similarly to the C++ standard auto_ptr construct [3]. We also suggest analogous treatment for Java arrays, featuring automatic selection of the access function based on the actual element type (e.g., Get<Int>ArrayElements), releasing array elements in the destructor of this "smart container", and conventional access to array elements with operator[]. The "construction as resource acquisition" idiom may also be applied to function pairs such as NewGlobalRef / DeleteGlobalRef used for reservation and release of global references (respectively), MonitorEnter / MonitorExit used for protecting critical sections of code etc.

To make the discussion concrete, we start with developing a running example which solves a toy problem. We first show the native code for solving this task without using the framework ("before"), and then show the desired code ("after"), streamlined using the proposed framework. In a subsequent section we develop the JNI encapsulation framework step by step. The article ends with a larger-scale example. The entire code for this article with complete Java – C++ integration examples can be obtained from the C++ Report Web site at http://www.creport.com.

Running example

Our toy problem defines a Java class JniExample with an integer, a static String and an integer array fields. Function main calls a native function implemented in C++, that accesses the Java fields, prints their original (default) values, and ultimately modifies them. Listing 1 shows (a fragment of) the Java class whose variables need to be accessed from C++ code.

Listing 1. A Java class for the running example (excerpt from JniExample.java):

```
public class JniExample {
  public int intField = 17;                          // integer field
  public static String stringField = "Hello, world!"; // static String field
  public int[] intArray = new int[2];                // integer array
  …
  private static native void native_call(JniExample x); // sample native call
  …
}
```

Listing 2 shows sample implementation of native C++ code that modifies the above variables, implemented using the original JNI [1]. Observe that on average 3 to 4 preparatory operations are necessary to access a Java field from the native code. The revised code in Listing 3 reduces this overhead to one constructor invocation per access. Note also that assignment to Java fields becomes more intuitive too.

Listing 2. Sample native code that uses the original JNI (excerpt from jni_example_org.cpp):

```
JNIEXPORT void JNICALL Java_JniExample_org_1native_1call (JNIEnv *env, jclass clazz, jobject obj) {
  // Lookup the integer field ('intField') in 'obj' and get its value
  jfieldID intFieldId = env->GetFieldID(clazz, "intField", "I");
  if (id == 0) throw JNIException("Field not found");
  int intFieldVal = env->GetIntField(obj, intFieldId);

  // Lookup the static String field ('stringField') in 'obj', then convert the Java string representation
  // to C++ type 'const char *'.
  jfieldID strFieldId = env->GetStaticFieldID(clazz, "stringField", "Ljava/lang/String;");
  if (id == 0) throw JNIException("Field not found");
  jstring jstr = (jstring) env->GetStaticObjectField(clazz, strFieldId);
  const char *str = (jstr == 0) ? 0 : env->GetStringUTFChars(jstr, 0);

  // Lookup the integer array field ('intArray') in 'obj', then convert it to C++ type 'jint *'.
  jfieldID arrFieldId = env->GetFieldID(clazz, "intArray", "[I");
  if (id == 0) throw JNIException("Field not found");
  jintArray jarr = (jintArray) env->GetObjectField(obj, arrFieldId);
  jint *arr = env->GetIntArrayElements(jarr, 0);

  // Set new values
  env->SetIntField(obj, intFieldId, 0);
  arr[0] = 0; arr[1] = 0;
  env->SetStaticObjectField(clazz, strFieldId, env->NewStringUTF("Good-bye, world!"));

  // Explicitly release resources
  env->ReleaseIntArrayElements(jarr, arr, 0);
  env->ReleaseStringUTFChars(jstr, str);
}
```

Listing 3. Sample native code that uses the JNI encapsulation framework (excerpt from jni_example.cpp):

```
JNIEXPORT void JNICALL Java_JniExample_native_1call (JNIEnv *env, jclass clazz, jobject obj) {
  // Lookup the Java fields in 'obj'
  JNIField<jint> intField(env, obj, "intField");
  JNIStringUTFChars str(env, "JniExample", "stringField");
  JNIArray<jint> arr(env, obj, "intArray");

  // Set new values
  intField = 0;
  arr[0] = 0; arr[1] = 0;
  JNIStaticField<jstring>(env, obj, "stringField") = env->NewStringUTF("Good-bye, world!");

  // Destructors for 'arr' and 'str' are invoked automatically
}
```

Template-based encapsulation of the JNI

In this section we evolve the JNI encapsulation framework. First, we discuss access to scalar variables (of both primitive and user-defined types). We then develop a generic resource management scheme which underlies the implementation of containers (arrays and strings). Finally, we apply this scheme to advanced JNI features such as monitors and global references.

Field access

Accessing a Java variable from C++ is a cumbersome process at the least. Our aim is to develop a technique for establishing correspondence between a C++ object and the Java variable, so that all low-level access operations become transparent to the programmer. For example, a C++ object corresponding to intField is a proxy[2] of type JNIField<jint>, created using the environment handle and the Java object passed via the JNI native call: JNIField<jint> intField(env, obj, "intField"). Then changing the variable value in C++ is as simple as that: intField = 17.

To this end, we use a template class JNIField (see Listing 4), whose three constructors present three ways to attach a C++ variable to a Java field. The first constructor receives an environment handle, a reference to the Java object whose field is to be accessed, the field name and type signature, and connects to the designated field. The second constructor allows creation of fields whose type signature may be deduced automatically (more on this below). Both constructors compute the field identifier (using implicit calls to the JNI functions GetObjectClass and GetFieldId; see Listing 5), and store it together with the object handle. The assignment and casting operators provide easy access to the variable value.

Occasionally, it might be necessary to access the same field in numerous Java objects of the same type (e.g., iterating over array elements). In such a case, the field identifier may be computed only once, and cached for subsequent reuse; this mode is supported by the third constructor.

Listing 4. JNIField template (excerpt from jni_field.h):

```
template<class NativeType> class JNIField {
  typedef JNIField<NativeType> _self;
  jobject _obj;                          // Object that holds the field
  JNIFieldId<NativeType> _id;            // field ID

public:
  JNIField(JNIEnv *env, jobject obj, const char *name, const char *sig) :
          _obj(obj), _id(env, obj, name, sig) {}
  JNIField(JNIEnv *env, jobject obj, const char *name) :
          _obj(obj), _id(env, obj, name) {}
  JNIField(JNIFieldId<NativeType> id, jobject obj) : _obj(obj), _id(id) {}

  _self &operator= (const _self &rhs);
  _self &operator= (const NativeType &rhs);
```

[2] The name of the C++ variable is obviously arbitrary; we use the same variable name in C++ and Java merely for convenience.

```
  // Casting to NativeType
  operator NativeType() const { return _id.Get(_obj); }
};
```

A base class JNIGenericFieldId defines a generic field identifier, used to build both regular and static field identifiers (JNIFieldId and JNIStaticFieldId, respectively). It contains a pointer to the JNI environment (of type JNIEnv*) and a field identifier (of type jfieldID). Listing 5 shows the class definitions (for brevity sake, JNIStaticFieldId is not shown; it is largely similar to JNIFieldId, except it uses GetStaticFieldId instead of GetFieldId).
JNIFieldId is actually a class template, which may be instantiated for actual data types (through the JavaType parameter). In its turn, the class constructor is realized as a *member template*. Its protoClass parameter facilitates the various ways to determine the class whose field is being accessed. A class type may be either specified directly, or computed from a class instance (i.e., the object itself) or the class name. Class JNIClass (Listing 6) encapsulates this underlying functionality, implementing all the possible ways to compute a Java class type. JNIFieldId template also defines a pair of generic field access functions Get and Set, which are subsequently specialized for primitive types.

Listing 5. Field identifier (excerpt from jni_field.h):

```
class JNIGenericFieldId {
protected:
  JNIEnv *_env;  // Environment handle for subsequent field manipulation
  jfieldID _id;  // Field ID

  JNIGenericFieldId(JNIEnv *env, jfieldID id) : _env(env), _id(id)
    { if (_id == 0) throw JNIException("Field not found"); }
};

template<class JavaType> class JNIFieldId : public JNIGenericFieldId {
public:
  template<class T>
  JNIFieldId(JNIEnv *env, T protoClass, const char *name,
             const char *sig = SIGNATURE_OF(JavaType)) :
    JNIGenericFieldId(env, env->GetFieldID(JNIClass(env, protoClass), name, sig)) {}

  JavaType Get(jobject obj) const
    { return static_cast<JavaType>(_env->GetObjectField(obj, _id)); }
  void Set(jobject obj, JavaType val)
    { _env->SetObjectField(obj, _id, val);}
};
```

Listing 6. JNIClass – all the ways to construct a class (jni_class.h):

```
class JNIClass {
  jclass _clazz;       // class handle
public:
  JNIClass(JNIEnv *env, jclass clazz) : _clazz(clazz) {}
  JNIClass(JNIEnv *env, jobject obj) : _clazz(env->GetObjectClass(obj))
    { if (_clazz == 0) throw JNIException("Failed to get a class"); }
  JNIClass(JNIEnv *env, const char *name) : _clazz(env->FindClass(name))
    { if (_clazz == 0) throw JNIException("Failed to get a class"); }
 JNIClass(jclass clazz) : _clazz(clazz) {}

  // Casting operators
```

```
  operator jclass() { return _clazz; }
  operator const jclass() const { return _clazz; }
};
```

An interesting aspect of the field identifier template (JNIFieldId) is its type inference capability, which is mostly hidden from the user. Observe that the JNI only supplies a set of Get<type>Field functions, each corresponding to a different variable type; thus, accessing a variable apparently requires explicit specification of the appropriate function. We circumvent this requirement by using the *template specialization* technique, namely, we preinstantiate the member functions of JNIFieldId for all the primitive types (i.e., JNIFieldId<jint>::operator=() is actually implemented using the SetIntField function etc.). This way, once the field is instantiated (for example, on an integer Java variable – JNIField<jint>), we no longer need to specify its type on every variable access.

To achieve this aim, we start with a series of basic declarations, where each primitive type is associated with the corresponding Java basic type and Java array type, and is assigned the JNI type signature. The declarations are realized as C++ structs, which are later used by the compiler as a lookup table (among the rest, this relieves the user from having to remember signatures of various Java types; instead, those can be looked up whenever necessary). Listing 7 shows such definitions for the integer primitive type:

Listing 7. Basic declarations (excerpt from jni_declarations.h):

```
struct IntDeclarations {
  typedef jint NativeType;
  typedef jintArray ArrayType;
  static const char *signature() { return "I"; }
  static const char *array_signature() { return "[I"; }
};
```

The following macros[3] retrieve type-specific declarations from such structures:

```
#define NATIVE_TYPE(Type)          Type##Declarations::NativeType
#define ARRAY_TYPE(Type)           Type##Declarations::ArrayType
#define SIGNATURE(Type)            Type##Declarations::signature()
#define ARRAY_SIGNATURE(Type)      Type##Declarations::array_signature()
```

For example, the Java type corresponding to the integer primitive type is NATIVE_TYPE(Int) = IntDeclarations::NativeType = jint. It is these macros that automatically map the Int part of the SetIntField function name into the jint variable type (with similar treatment for other types; see the usage in Listing 10).

Finally, we employ the C++ preprocessor to empower the compiler with knowledge of which function to use in each context. Listing 8 shows a macro block that looks up basic type definitions to deduce which function to use.

Listing 8. Macro block with definitions of JNIField member functions (excerpt from jni_field.h):

```
#define JNI_FIELD_ID_METHODS(Type)                                          \
template<> class JNIFieldId<NATIVE_TYPE(Type)> : public JNIGenericFieldId {  \
```

[3] ## is the concatenation operator of the C++ preprocessor.

```
public:                                                                    \
  template<class T> JNIFieldId(JNIEnv *env, T protoClass, const char *name) :  \
          JNIGenericFieldId(env, env->GetFieldID(JNIClass(env, protoClass),  \
                                                 name, SIGNATURE(Type))) {}  \
                                                                           \
  NATIVE_TYPE(Type) Get(jobject obj) const                                 \
    { return _env->Get##Type##Field(obj, _id); }                           \
  void Set(jobject obj, NATIVE_TYPE(Type) val)                             \
    { _env->Set##Type##Field(obj, _id, val); }                             \
};
```

To instantiate this macro for the primitive types, we invoke the macro INSTANTIATE_FOR_PRIMITIVE_TYPES(JNI_FIELD_ID_METHODS), defined in file jni_declarations.h as follows:

```
#define INSTANTIATE_FOR_PRIMITIVE_TYPES(BLOCK_MACRO) \
BLOCK_MACRO(Boolean)                                 \
BLOCK_MACRO(Byte)                                    \
BLOCK_MACRO(Char)                                    \
BLOCK_MACRO(Short)                                   \
BLOCK_MACRO(Int)                                     \
BLOCK_MACRO(Long)                                    \
BLOCK_MACRO(Float)                                   \
BLOCK_MACRO(Double)
```

Let us trace step by step what happens during compilation, using the integer type as an example. When the compiler expands the macro block for Int, it instantiates (more exactly, specializes) the JNIFieldId template with the parameter NATIVE_TYPE(Int) = jint. Listing 9 presents the (formatted) preprocessor output for JNIFieldId<jint> member functions. Thus, the member functions are realized in terms of correct accessor functions for the Int type.

Listing 9. Preprocessor output for expanding JNIField member functions

```
template<> class JNIFieldId<IntDeclarations::NativeType> : public JNIGenericFieldId {
public:
  template<class T> JNIFieldId(JNIEnv *env, T protoClass, const char *name) :
    JNIGenericFieldId(env, env->GetFieldID(JNIClass(env, protoClass),
                                           name, IntDeclarations::signature())) {}

  IntDeclarations::NativeType Get(jobject obj) const
    { return _env->GetIntField(obj, _id); }
  void Set(jobject obj, IntDeclarations::NativeType val)
    { _env->SetIntField(obj, _id, val); }
};
```

As explained above, the operators of JNIField are defined in terms of Get/Set functions of JNIFieldId:

```
JNIField<NativeType> &operator= (const NativeType &rhs) {
   _id.Set(_obj, rhs);
   return *this;
}

operator NativeType() const   // casting to NativeType
```

```
{ return _id.Get(_obj); }
```

To conclude the presentation of field access, file jni_field.h has similar definitions for static fields ("class variables" in Java terminology). It defines a class template JNIStaticFieldId, which inherits from JNIGenericFieldId and is specialized for the primitive types, and a class template JNIStaticField, which has an additional data member of type jclass for keeping the appropriate class object.

Array type declarations

File jni_declarations.h also provides a set of declarations to facilitate JNI arrays. We actually build a lookup table, which the compiler consults for type inference. This table may be looked up either given a primitive type or an array type. For example, this way the compiler can automatically deduce that an array of jints is of type jintArray, and that an array of type jcharArray consists of jchars and has the signature "[C". The template specialization technique is used here again, to duplicate basic declarations for array types (see Listing 10).

Listing 10. Declarations for regular types and array types (excerpt from jni_declarations.h):

```
template<class JavaType> struct JNITypeDeclarations {};

#define JNI_TYPE_DECLARATIONS(Type)                                    \
template<> struct JNITypeDeclarations<NATIVE_TYPE(Type)> {             \
  typedef Type##Declarations Declarations;                             \
  typedef NATIVE_TYPE(Type) NativeType;                                \
  typedef ARRAY_TYPE(Type) ArrayType;                                  \
  static const char *signature() { return SIGNATURE(Type); }           \
};
#define JNI_ARRAY_DECLARATIONS(Type)                                   \
template<> struct JNITypeDeclarations<ARRAY_TYPE(Type)> {              \
  typedef Type##Declarations Declarations;                             \
  typedef NATIVE_TYPE(Type) NativeType;                                \
  typedef ARRAY_TYPE(Type) ArrayType;                                  \
  static const char *signature() { return ARRAY_SIGNATURE(Type); }     \
};

// Combo instantiation of JNITypeDeclarations specializations for all primitive types
INSTANTIATE_FOR_PRIMITIVE_TYPES(JNI_TYPE_DECLARATIONS)
INSTANTIATE_FOR_PRIMITIVE_TYPES(JNI_ARRAY_DECLARATIONS)

// auxiliary macros for mapping JNI types into corresponding declarations
#define DECLARATIONS_OF(JavaType)       \
JNITypeDeclarations<JavaType>::Declarations
#define NATIVE_TYPE_OF(JavaType)        \
JNITypeDeclarations<JavaType>::NativeType
#define ARRAY_TYPE_OF(JavaType)         \
JNITypeDeclarations<JavaType>::ArrayType
#define SIGNATURE_OF(JavaType)          \
JNITypeDeclarations<JavaType>::signature()
```

Listing 11 shows the expanded macro block with specialization of the JNITypeDeclarations template for jint and jintArray (formatted preprocessor output).

Listing 11. Sample preprocessor output for expanding the specialization of the JNITypeDeclarations structure

```
template<> struct JNITypeDeclarations<IntDeclarations::NativeType>
{
  typedef IntDeclarations Declarations;
  typedef IntDeclarations::NativeType NativeType;
  typedef IntDeclarations::ArrayType  ArrayType;
  static const char *signature()
  { return IntDeclarations::signature(); }
};

template<> struct JNITypeDeclarations<IntDeclarations::ArrayType>
{
  typedef IntDeclarations Declarations;
  typedef IntDeclarations::NativeType NativeType;
  typedef IntDeclarations::ArrayType  ArrayType;
  static const char *signature()
  { return IntDeclarations::array_signature(); }
};
```

User-defined data types

Our framework also allows additional declarations for custom (non-primitive) types, so that they can be used by the compiler for automatic type inference. Listing 12 exemplifies this feature with the declarations structure for String data type.

Listing 12. String as an example of a custom data type (excerpt from jni_declarations.h):

```
struct StringDeclarations {
   typedef jstring NativeType;
   typedef jobject ArrayType;
   static const char *signature() { return "Ljava/lang/String;"; }
   static const char *array_signature() { return "[Ljava/lang/String;"; }
};

JNI_TYPE_DECLARATIONS(String)
```

Such declarations are immediately available for the compiler to utilize. For example, to assign a new value to the string field of the running example (see Listing 1) from the C++ code, we use the following definition:
JNIStaticField<jstring>(env, obj, "stringField") = env->NewStringUTF("Good-bye, world!");
Specifically, we do not specify explicitly the corresponding type signature. The compiler infers it automatically, due to the default value const char *sig = SIGNATURE_OF(JavaType) in the constructor of JNIStaticFieldId (a similar case for JNIFieldId is shown in Listing 5).

Invocation of Java methods

Java methods may be invoked from native code using dedicated JNI functions Call<Type>Method (as well as Call<Type>MethodA and Call<Type>MethodV), with explicit specification of the types of parameters and the return value. Apparently, a similar template trick could be used to automate these function calls too. However, this happens to be quite difficult on a more detailed examination, as C++ performs no type inference

on function return values (and anyway, complex parameter signatures would be very hard to automate).

Resource management

In this section we develop a general resource management mechanism, and then apply it to simplify various JNI use cases. Our resource management approach is based on the C++ "construction as resource acquisition" idiom [2]: resources are allocated in the constructor, and are released in the destructor of dedicated auxiliary objects. This idiom is implemented using the Proxy pattern [4], with functionality similar to that of auto_ptr template of the C++ Standard Library [3].

Listing 13 shows a JNIResource template, whose parameter is assumed to provide the following four definitions:

- JResource type: the original Java resource type (e.g., jintArray)
- Resource type: the corresponding exported resource (e.g., jint*)
- GetF: functional object for resource allocation, of the form
  Resource GetF::operator()(JNIEnv *, JResource)
- ReleaseF: functional object for resource deallocation, of the form
  void ReleaseF::operator()(JNIEnv *, JResource, Resource)

The GetF and ReleaseF functional objects provide default allocation and deallocation facilities. Note that the JNI allows customized resource management (e.g., the isCopy parameter for allocation and the mode parameter for deallocation). This behavior could be achieved by supplying a user-defined functional object as an additional parameter to the constructor (in case of allocation), or to the function ReleaseResource (to be used prior to destruction for explicit customized deallocation).

Listing 13. JNIResource template (excerpt from jni_resource_base.h; trivial function bodies have been omitted for brevity):

```
template<class JNIResourceSettings> class JNIResource {
  typedef JNIResource<JNIResourceSettings> _self;
  typedef typename JNIResourceSettings::JResource JResource;
  typedef typename JNIResourceSettings::Resource Resource;
  typedef typename JNIResourceSettings::GetF DefaultGetF;
  typedef typename JNIResourceSettings::ReleaseF DefaultReleaseF;

  bool _owns;                 // true if the current object owns the resource
protected:
  JNIEnv *_env;               // Java environment handle
  JResource _jresource;       // Java resource handle
  Resource _resource;         // resource handle

public:
  JNIResource() : _env(0), _jresource(0), _resource(0), _owns(0) {}
  JNIResource(JNIEnv *env, JResource jresource) :
    _env(env), _owns(true), _jresource(jresource)
  { _resource = DefaultGetF()(_env, _jresource); }

  template<class GetF>
  JNIResource(JNIEnv *env, JResource jresource, GetF &getF) :
    _env(env), _owns(true), _jresource(jresource)
```

```
  { _resource = getF(_env, _jresource); }

  template<class GetF>
  JNIResource(JNIEnv *env, JResource jresource, const GetF &getF);

  JNIResource(_self &x) : _env(x._env), _owns(x._owns),
                          _jresource(x._jresource), _resource(x.release()) {}
  _self &operator= (JNIResource &x);

  ~JNIResource() { ReleaseResource(); }

  void ReleaseResource()
  { if (_owns) DefaultReleaseF()(_env, _jresource, release()); }

  template<class ReleaseF>
  void ReleaseResource(ReleaseF &releaseF)
  { if (_owns) releaseF(_env, _jresource, release()); }

  template<class ReleaseF>
  void ReleaseResource(const ReleaseF &releaseF);

  // casting operators
  operator Resource() { return get(); }
  operator const Resource() const { return get(); }

  Resource &get() { return _resource; }
  const Resource &get() const { return _resource; }

  Resource release();
};
```

We now proceed to a number of JNI use cases, and demonstrate how they can be simplified with the resource management described above. In these cases, JNIResource template is used as a base class, from which individual resource managers for strings, arrays etc. are derived. To instantiate the template, each resource managers defines an auxiliary structure of settings, which provides the four mandatory components of the template parameter (see above). Notice that all the resources that inherit from JNIResource have a default constructor, so that arrays of resources can be defined.

Strings

The JNI allows two kinds of strings, namely, using regular (UTF-8) or wide (Unicode) characters. Listing 14 exemplifies the former case. The C++ Resource of type const char* corresponds to the original Java resource of type jstring (JResource in our terms). JNIStringUTFCharsSettings implements (what in Java terminology would be called interface) JNIResourceSettings, which can serve a parameter to JNIResource template above. Applications should use class JNIStringUTFChars, which inherits from JNIResource<JNIStringUTFCharsSettings> and provides asString conversion function (for convenient usage of C++ std::string instead of raw char*; note, however, that such conversion physically copies the characters).

The constructors of JNIStringUTFChars use the default GetF allocator to acquire the string characters (via the underlying JNI function GetStringUTFChars), and the destructor

releases them via ReleaseStringUTFChars. If isCopy parameter is in use, its value is updated to reflect if original Java string characters have been copied to temporary storage. The first two (non-default) constructors require explicit specification of the Java resource (JResource), while the last two compute it "on the fly", by first building a JNIField (or a JNIStaticField) and then obtaining its handle. This technique[4] allows to easily access string fields of objects given an object handle, its class type or class name (for static fields). For instance, the following code attaches a C++ variable to the static string field of the running example (see Listing 1), using the names of the host class and the string field: JNIStringUTFChars str(env, "JniExample", "stringField"). The string value can then be printed simply using cout << str.get().
Observe that as Java strings cannot be modified, such are also the strings exported into C++. Hence, only the const version of operator[] is provided here. Function length returns the length of the C++ string (using the JNI function GetStringUTFLength).

Listing 14. Accessing string characters as UTF-8 (excerpt from jni_resource.h):

```
struct JNIStringUTFCharsSettings {
  typedef jstring JResource;
  typedef const char *Resource;

  struct GetF {
    jboolean *_isCopy;
    GetF(jboolean *isCopy = 0) : _isCopy(isCopy) {}
    Resource operator() (JNIEnv *env, JResource jstr) const
    { return env->GetStringUTFChars(jstr, _isCopy); }
  };

  struct ReleaseF {
    void operator() (JNIEnv *env, JResource jstr, Resource str) const
    { env->ReleaseStringUTFChars(jstr, str); }
  };
};

class JNIStringUTFChars : public JNIResource<JNIStringUTFCharsSettings> {
  typedef JNIStringUTFCharsSettings _settings;
  typedef JNIResource<_settings> _super;
public:
  JNIStringUTFChars() {}
  JNIStringUTFChars(JNIEnv *env, jstring jstr) : _super(env, jstr) {}
  JNIStringUTFChars(JNIEnv *env, jstring jstr, jboolean *isCopy) :
    _super(env, jstr, _settings::GetF(isCopy)) {}

  template<class T>
  JNIStringUTFChars(JNIEnv *env, T arg, const char *name);
  template<class T>
  JNIStringUTFChars(JNIEnv *env, T arg, const char *name, bool isStatic);

  const char &operator[] (int i) const { return _resource[i]; }
  const int length() const { return env->GetStringUTFLength(jresource); }
  string asString() const { return string(_resource); }
};
```

[4] Such computation of the Java resource is performed by an auxiliary template class GetJResource; for the complete definition of this mechanism see file jni_resource.h.

To replace the value of a Java string, we should first create a new string that can later "survive" in the Java environment. In the code fragment below, a C++ variable is instantiated on the static string field of the running example (see Listing 1), and then assigned a brand new Java string created with the JNI function NewStringUTF:
JNIStaticField<jstring>(env, obj, "stringField") = env->NewStringUTF("Good-bye, world!");
Note that there is no need to worry about eventually releasing the memory occupied by this newly created string – this is performed by Java garbage collector.

File jni_resource.h also defines class JNIStringChars for accessing Java strings with wide (Unicode) characters. This definition is mostly similar to JNIStringUTFChars, except it uses a Resource of type const jchar* instead of const char*.

Arrays

Arrays feature most of the functionality presented till now. In particular, they provide automatic acquisition and release of elements in the constructor and destructor respectively (the template specialization trick is used here again to preinstantiate the array template for all primitive types, so that the appropriate Get<Type>ArrayElements / Release<Type>ArrayElements functions are automatically selected based on the context). Function size uses the JNI facility GetArrayLength to determine the number of array elements. Two versions of operator[] (regular and const) are provided to access the individual elements.

File jni_resource.h contains the implementation. Template class JNIArray inherits from JNIResource, parameterized with the appropriate JNIArraySettings. The latter is a template in its own right, which has a parameter specifying the native element type. The array type and signature are obtained from the element type using the declarations of file jni_declarations.h (see Listing 10). To access the integer array field of the running example (see Listing 1), we instantiate a corresponding C++ variable as follows:
JNIArray<jint> arr(env, obj, "intArray"). Subsequent access of the array elements is straightforward: arr[0] = 0.
The default behavior of the JNIArray is to copy all the array elements back into Java environment once the C++ array goes out of scope (this is done by function Release<Type>ArrayElements invoked from the array destructor). When this behavior needs to be overridden, use member function CustomRelease to set the desired mode for Release<Type>ArrayElements.

Occasionally, it is not necessary to manipulate an entire Java array, which may be quite large. For cases when only a part of the array needs to be accessed, the JNI provides a pair of functions Get<Type>ArrayRegion / Set<Type>ArrayRegion. File jni_utils.h defines template functions GetArrayRegion / SetArrayRegion (preinstantiated at compile time for primitive types) that are capable of deducing the element type based on their parameters.

Monitors

Monitors serve to ensure mutual exclusion of threads competing for a shared resource. To ensure resource integrity ("thread safety"), threads should request to enter a monitor at

the beginning of the critical section, and leave it at the end of the section. We propose a resource management technique that uses an auxiliary automatic object of type JNIMonitor, whose constructor enters a monitor, and whose destructor leaves the monitor as soon as the object goes out of scope (see Listing 15). The constructor of JNIMonitor receives a handle to the object that constitutes a shared resource protected by this monitor.

Listing 15. Monitors (excerpt from jni_resource.h):

```
struct JNIMonitorSettings {
  typedef jobject JResource;
  typedef jobject Resource;

  struct GetF {
    Resource operator() (JNIEnv *env, JResource obj) const {
      env->MonitorEnter(obj);
      return obj;
    }
  };
  struct ReleaseF {
    void operator() (JNIEnv *env, JResource obj, Resource dummy) const
    { env->MonitorExit(obj); }
  };
};

class JNIMonitor : public JNIResource<JNIMonitorSettings> {
public:
  JNIMonitor() {}
  JNIMonitor(JNIEnv *env, jobject obj) :
    JNIResource<JNIMonitorSettings>(env, obj) {}
};
```

Here is a sample code fragment which uses a monitor:

```
void sample_function(JNIEnv *env, jobject obj) {
...
 { // start the critical section block
   JNIMonitor mon(env, obj);
   // do the critical section stuff here
   ...
   // the destructor of 'mon' automatically exits
   // the underlying Java monitor
 }
...
}
```

Global references

It is sometimes necessary to obtain a reference to a Java object, so that it can be used across the function boundaries of C++ code. In such cases, a *global reference* to this object should be reserved (in contrast to most JNI functions that yield a *local* reference, which expires as soon as the current scope terminates). The last use case defined in file jni_resource.h implements an auxiliary template class JNIGlobalRef, whose constructor acquires and destructor releases a global reference to the specified object (see sample usage in Listing 16).

Using the code

Following the STL methodology, all the framework code resides in header files and is entirely template based, so clients do not need to compile their applications with any additional libraries. In fact, client applications only need to include the master file jni_master.h, which itself #includes all the other headers. The entire code of this article with complete Java – C++ integration examples can be obtained from the C++ Report Web site at http://www.creport.com.

A more elaborate example

We now apply the JNI encapsulation framework to a more substantial example. Suppose we have a Java application in which several concurrent threads generate (a predefined number of) objects with string IDs. The task is to collect these objects in a thread-safe way, and ultimately sort them by their IDs. Since earlier versions of the Java Development Kit (such as JDK 1.1) did not have the Collections Framework, it is quite natural to implement the sorting container in native C++ code using the STL. File JniComplexExample.java (not shown here for the sake of brevity) contains the Java part of this example, which uses the following native functions:

- init_native_resources() – initializes the native code data structures
- clean_native_resources() – releases the native resources
- register_object() – inserts a given object into the container
- recall_objects() – returns a sorted array of the collected objects

Listing 16 shows the native code for the container, implemented using the STL multimap. The container holds the (Java originated) objects by global references[5], associated with their string IDs. The container is realized as a singleton object[6], which has two thread-safe access functions:

- insert() – collects a given object
- exportAllObjects() – returns all the collected objects as a vector (observe that this vector is inherently sorted, as the objects are extracted from a multimap).

To ensure code portability, thread-safety is implemented using JNI monitors. Critical sections start with monitor definition and last until the monitor is automatically destroyed as it goes out of scope.

Listing 16. Native code implementation of a thread-safe sorting container (excerpt from file jni_complex_example.cpp):

```
class SampleContainer {
  friend class auto_ptr<SampleContainer>;
  static auto_ptr<SampleContainer> instance;

  typedef multimap<string, JNIGlobalRef<jobject> *> MapOfObjects;
  MapOfObjects mapOfObjects;          // the container implementation
  JNIGlobalRef<jobject> monitor;      // monitor (for critical sections)
  JNIEnv *_env;                       // the environment variable

  SampleContainer(JNIEnv *env) : _env(env), monitor(env, getMonitorObject(env)) {}
```

[5] Global references are required so that the Java garbage collector does not destroy the objects prematurely.
[6] Our singleton implementation uses the STL auto_ptr along the guidelines of [5].

```
  ~SampleContainer();                    // the dtor purges all the map elements

  jobject getMonitorObject(JNIEnv *env) {  // allocating the monitor object
   JNIClass objectClass(env, "java/lang/Object");
   jmethodID constructorId = env->GetMethodID(objectClass, "<init>", "()V");
   return env->NewObject(objectClass, constructorId);
  }

public:
  static SampleContainer *getInstance(JNIEnv *env = 0) {
    if (instance.get() == 0) {
      if (env == 0) {     // raise an exception flag in Java, then throw a C++ exception
        env->ThrowNew(JNIClass(env, "java/lang/Exception"), “Init failed”);
        throw new JNIException("Init failed");
      }
      static JNIGlobalRef<jobject> initMonitor(env, getMonitorObject(env));

      // Double-checked locking is used to provide correct initialization
      JNIMonitor startCriticalSection(env, initMonitor);
      if (instance.get() == 0)
        instance = auto_ptr<SampleContainer>(new SampleContainer(env));
    }
    return instance.get();
  }

  static void clean() { delete instance.release(); }    // explicitly release the native resources

  void insert(jobject obj) {
    JNIMonitor startCriticalSection(_env, monitor);

    JNIStringUTFChars str(_env, obj, "name"); // retrieve the object ID
    JNIGlobalRef<jobject> *ref = new JNIGlobalRef<jobject>(_env, obj);
    mapOfObjects.insert(make_pair(str.asString(), ref));
   }

  vector<JNIGlobalRef<jobject> *> exportAllObjects() {
    JNIMonitor startCriticalSection(_env, monitor);

    vector<JNIGlobalRef<jobject> *> result(mapOfObjects.size(), 0);
    MapOfObjects::iterator p; vector<JNIGlobalRef<jobject> *>::iterator q;
    for (p = mapOfObjects.begin(), q = result.begin();
         p != mapOfObjects.end(); p++, q++)
      *q = (*p).second;
    return result;
  }
};

// Singleton instance
auto_ptr<SampleContainer> SampleContainer::instance;

/* Implementation of native calls (note that their prototypes are automatically generated) */

// init_native_resources()
JNIEXPORT void JNICALL
Java_JniComplexExample_init_1native_1resources (JNIEnv *env, jclass clazz)
{ SampleContainer::getInstance(env); }
```

```
// clean_native_resources()
JNIEXPORT void JNICALL
Java_JniComplexExample_clean_1native_1resources (JNIEnv *env, jclass clazz)
{ SampleContainer::clean(); }

// register_object()
JNIEXPORT void JNICALL
Java_JniComplexExample_register_1object (JNIEnv *env, jclass clazz, jobject obj)
{ SampleContainer::getInstance()->insert(obj); }

// recall_objects
JNIEXPORT jobjectArray JNICALL
Java_JniComplexExample_recall_1objects (JNIEnv *env, jclass clazz) {
  // obtain the vector of global references
  vector<JNIGlobalRef<jobject> *> allObjects = SampleContainer::getInstance()->exportAllObjects();
  // create an output array of type 'NameWithInfo[]'
  JNIClass objectClass(env, "NameWithInfo");
  jobjectArray result = env->NewObjectArray(allObjects.size(), objectClass, 0);
  // export the objects
  for (int i = 0; i < allObjects.size(); i++)
    env->SetObjectArrayElement(result, i, *allObjects[i]);
  return result;
}
```

Discussion

Let us review the properties of the solution we developed:

1. *Easier to use, more straightforward approach*
   Whenever possible, the compiler infers variable types from the context. Java data structures are automatically exported to (and in order to save changes, are later imported from) the C++ code. Auxiliary technical operations are encapsulated in higher-level templates.
2. *Less error-prone API*
   Fewer functions to call means fewer opportunities to err in successive function invocations. Also, it is now possible to perform various checks at compile time, instead of discovering problems much later as run-time errors.
3. *Proper resource management*
   Resources are automatically deallocated when they are no longer necessary, thus preventing resource leaks, deadlocks, and starvation.
4. *Portability issues*
   Java portability is preserved by using only ANSI-standard C++ and the STL [3].
5. *Compilation overhead*
   A possible drawback of the suggested framework is the compile-time penalty it imposes, due to heavy use of the preprocessor and embedded templates. However, this overhead is limited to the compilation time, and does not propagate to the run-time. The code size increase is also negligible, since most of the templates only provide type definitions (and thus do not need run-time representation at all), and unused template instantiations are discarded by the code optimizer.

Acknowledgments

The authors are thankful to Marc Briand, Herb Sutter and Jerry Schwarz for their constructive comments and suggestions.

References

[1] Java Native Interface Specification
http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html
[2] Stroustrup, B. "The C++ Programming Language", 3rd edition, Addison Wesley, 1990.
[3] "Information Technology – Programming Languages – C++",
International Standard ISO/IEC 14882-1998(E).
[4] Gamma, E., *et al.* "Design Patterns: Elements of Reusable Software Architecture",
Addison Wesley, 1995.
[5] Gabrilovich, E. "Destruction-Managed Singleton: A Compound Pattern for Reliable
Deallocation of Singletons", C++ Report, 12(3): 35-40, March 2000.

Sidebar: Java Native Interface in a nutshell

As set forth in [1], the *Java Native Interface (JNI)* enables Java code running on a Java Virtual Machine (JVM) to work in concert with code written in other programming languages (e.g., C, C++ or assembly, referred to as “native languages” due to their “nativeness” to the execution environment). The JNI can be handy to foster code reuse or to implement mission-critical parts of the code in a native language for superior efficiency. The JNI may also occasionally be useful to facilitate platform-dependent features not available through the standard Java class library. The Java Native Interface provides *bidirectional* cooperation between Java code and native code, so that Java methods may transparently invoke native routines, and vice versa. Additional functional wealth available to native applications includes manipulating Java objects, access to Java class information and run-time type checking facilities, dispatching Java exceptions, as well as convenient usage of Java thread synchronization mechanisms. Finally, the so-called *Invocation API* allows any native application to directly operate the Java Virtual Machine as a regular native object.

Every native function receives *a JNI interface pointer* through which it calls all the other JNI functions. For the sake of implementation flexibility, the interface pointer indirectly points to a table of pointers to JNI functions. Observe, therefore, that calling a native method always entails several dereference operations. Note also that the interface pointer is only valid in the current thread. This pointer is implicit in Java signatures of native methods, and constitutes the (explicit) first parameter in their native programming language. The JNI also prescribes the meaning of the second parameter to native methods. This parameter contains a reference to the host object (this) for *instance methods* (nonstatic functions), and a reference to the Java class object for *class methods* (static functions). Libraries of native functions are normally loaded dynamically at run-time, using the System.loadLibrary method. Name mangling conventions for native methods allow overloading, and are stipulated by the JNI Specification.

The JNI allows native code to access Java objects of both primitive types (e.g., int, char) and user-defined types. The JNI Specification associates each Java primitive type with an equivalent native type (for instance, the jfloat C++ native type corresponds to the Java float, and is implemented as a 32-bit variable). Native methods may receive Java objects as parameters. Retrieving data members of a compound parameter (including individual array entries), or creating new objects in the Java environment, is performed by calling appropriate JNI functions. The Java Virtual Machine keeps track of all the objects made available to the native code, so that they do not get garbage-collected while in use. Calling Java methods and raising exceptions from the native code is also accomplished through the variety of JNI functions.

About the authors

Evgeniy Gabrilovich is an Algorithm Developer at Zapper Technologies Inc. He holds an M.Sc. degree in Computer Science from the Technion – Israel Institute of Technology. His interests involve Computational Linguistics, Information Retrieval, Artificial Intelligence, and Speech Processing. He can be contacted at gabr@acm.org.

Lev Finkelstein is an Algorithm Developer at Zapper Technologies Inc., and is a Ph.D. student in Computer Science at the Technion – Israel Institute of Technology. His interests include Artificial Intelligence, Machine Learning, Multi-agent systems, and Data Mining. He can be reached at lev@zapper.com.