C and C++

8. JNI and STL

Andrew W. Moore

University of Cambridge
(with thanks to Alastair R. Beresford and Bjarne Stroustrup)

Michaelmas Term 2010
JNI — Java Native Interface

Java Native Interface (JNI) is the Java interface to non-Java code.

- interoperate with applications and libraries written in other programming languages (e.g., C, C++, assembly)

Examples we will look at:

- Embedding C in Java
- Using Java from C

Why use JNI?

- Low-level specifics not provided by Java
- Performance (Java is not famous for being fast)
- Legacy code wrapper
Justification

Pro:
- Reuse: allow access to useful native code
- Efficiency: use best language for the right task

Cons:
- Applets: mobility makes this painful
- Portability: native methods aren’t portable
- Extra work: javah, creating shared library
Embedding C in Java

1. Declare the method using the keyword native (without implementation)
2. (Make sure Java loads the required library)
3. Run javah to generate names & headers
4. Implement the method in C
5. Compile as a shared library

```java
class HelloWorld {
    public native void displayHelloWorld();

    static {
        System.loadLibrary("hello");
    }

    public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
    }
}
```
Generate JNI Header

Compile HelloWorld.java
$ javac HelloWorld.java

Create HelloWorld.h
$ javah HelloWorld
HelloWorld.h

1    #include <jni.h>
2    /* Header for class HelloWorld */
3    ifndef _Included_HelloWorld
4    define _Included_HelloWorld
5    ifdef __cplusplus
6      extern "C" {
7        */
8
9        */
10       * Class:      HelloWorld
11       * Method:    displayHelloWorld
12       * Signature: ()V
13       */
14      JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld
15          (JNIEnv *, jobject);
16
17      ifdef __cplusplus
18      }
19      endif
20      endif
HelloWorldImp.c

```c
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNICALL void JNICALL Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject obj)
{
    printf("Hello world!\n");
    return;
}
```
Create a Shared Library

1 class HelloWorld
2 {
3     ...
4         System.loadLibrary("hello");
5     ...
6 }

Compile the native code into a shared library:

(on pwf, other OS may require the -I option)

cc -shared HelloWorldImpl.c -o libhello.so
Run the Program

$ java HelloWorld

Sometimes:

Hello World!

Othertimes:

Exception in thread "main" java.lang.UnsatisfiedLinkError: no hello
    at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1698)
    at java.lang.Runtime.loadLibrary0(Runtime.java:840)
    at java.lang.System.loadLibrary(System.java:1047)
    at HelloWorld.<clinit>(HelloWorld.java:6)

Could not find the main class: HelloWorld. Program will exit.

Problems? set your LD_LIBRARY_PATH:
export LD_LIBRARY_PATH=. 
Primitive Types and Native Equivalents

Some examples:

```c
typedef unsigned char jboolean; // 8, unsigned
typedef unsigned short jchar; // 16, unsigned
typedef short jshort; // 16
typedef float jfloat; // 32
typedef double jdouble; // 64
```

Each Java language element must have a corresponding native equivalent

- Platform-specific implementation
- Generic programming interface
The JNIEnv *env argument

- passed to every native method as the first argument.
- contains function entries used by native code.
- organized like a C++ virtual function table.

In C: (*env)->(env, foo);
In C++: env->(foo);

Through JNI interface pointer, native code can:

- operate on classes (loading, checking, etc.)
- catch and throw exceptions
- access fields
- call methods
- manipulate strings and arrays
- enter/exit monitors
Accessing Java Strings

The jstring type is not equivalent to the C string type

```c
/* Wrong way */
JNIEnv *env, 
jobject obj, jstring prompt)
{
printf("%s", prompt); ... 
}

/* Right way */
JNIEnv *env, 
jobject obj, jstring prompt)
{
    char buf[128];
    const char *str = (*env)->GetStringUTFChars(env, prompt, 0);
    printf("%s", str);
    /* release the memory allocated for the string operation */
    (*env)->ReleaseStringUTFChars(env, prompt, str); ... 
}
```
Accessing Java Array

/* Wrong way */
JNIEXPORT jint JNICALL Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr) {
  int i, sum = 0;
  for (i=0; i<10; i++) {
    sum += arr[i];
  } ...
/* Right way */
JNIEXPORT jint JNICALL Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr) {
  int i, sum = 0;
  /* 1. obtain the length of the array */
  jsize len = (*env)->GetArrayLength(env, arr);
  /* 2. obtain a pointer to the elements of the array */
  jint *body = (*env)->GetIntArrayElements(env, arr, 0);
  /* 3. operate on each individual elements */
  for (i=0; i<len; i++) { sum += body[i]; }
  /* 4. release the memory allocated for array */
  (*env)->ReleaseIntArrayElements(env, arr, body, 0);
Accessing Java Member Variables

class FieldAccess {
    static int si; /* signature is "si" */
    String s;    /* signature is "Ljava/lang/String;"; */
} /* run javap -s -p FieldAccess to get the signature */

    /* 1. get the field ID */
    fid = (*env)->GetStaticFieldID(env, cls, "si", "I");
    /* 2. find the field variable */
    si = (*env)->GetStaticIntField(env, cls, fid);
    /* 3. perform operation on the primitive*/
    (*env)->SetStaticIntField(env, cls, fid, 200);

    /* 1. get the field ID */
    fid = (*env)->GetFieldID(env, cls, "s", "Ljava/lang/String;" );
    /* 2. find the field variable */
    jstr = (*env)->GetObjectField(env, obj, fid);
    /* 3. perform operation on the object */
    jstr = (*env)->NewStringUTF(env, "123");
    (*env)->SetObjectField(env, obj, fid, jstr);
Calling a Java Method

1. Find the class of the object
   Call GetObjectClass

2. Find the method ID of the object
   Call GetMethodID, which performs a lookup for the Java method in a given class

3. Call the method
   JNI provides an API for each method-type
   e.g., CallVoidMethod(), etc.
   You pass the object, method ID, and the actual arguments to the method (e.g., CallVoidMethod)

   ```java
   jclass cls = (*env)->GetObjectClass(env, obj);
   jmethodID mid = (*env)->GetMethodID(env, cls, "hello", "(I)V");
   (*env)->CallVoidMethod(env, obj, mid, parm1);
   ```
Garbage Collection & Thread Issues

- Arrays and (explicit) global objects are *pinned* and must be explicitly released
- Everything else is released upon the native method returning

(JNIEnv *) is *only* valid in the current thread

- Interface pointer are not valid to pass between threads
- Local references are not valid to pass between threads
- Thread access to global variables requires locking
Synchronization

- Synchronize is available as a C call
- Wait and Notify calls through JNIEnv do work and are safe to use

In Java:
```java
synchronized(obj) {
    ... /* synchronized block */
    ... }
```

In C:
```c
(*env)->MonitorEnter(env, obj);
/* synchronized block */
(*env)->MonitorExit(env, obj);
```
Embedding a JVM in C

- JDK ships JVM as a shared library
- This is handled as a special function-call from C
- Call does not return
- The call provides a pointer for access from native code
JVMinC.c

#include <stdlib.h>
#include <jni.h>

int main(int argc, char *argv[]) {
    JNIEnv *env;
    JavaVM *jvm;
    jint res;
    jclass cls;
    jmethodID mid;
    jstring jstr;
    jobjectArray args;
    JavaVMInitArgs vm_args;
    JavaVMOption options[4];
    
    /* disable JIT */
    options[0].optionString = "-Djava.compiler=NONE";
    
    /* set user class and native library path */
    options[1].optionString = "-Djava.class.path=.";
    options[2].optionString = "-Djava.library.path=.";
    
    /* print JNI-related messages */
    options[3].optionString = "-verbose:jni";
```c
vm_args.version = JNI_VERSION_1_4;
vm_args.options = options;
vm_args.nOptions = 4;
vm_args.ignoreUnrecognized = 1;
/* spawn JVM, find class and the class entry-point */
res = JNI_CreateJavaVM(&jvm, (void **)&env, &vm_args);
cls = (*env)->FindClass(env, "JVMinCTest");
mid = (*env)->GetStaticMethodID(env, cls, "main", 
    "([Ljava/lang/String;)V");
/* create a valid string to pass */
jstr = (*env)->NewStringUTF(env, " from C!");
args = (*env)->NewObjectArray(env, 1, 
    (*env)->FindClass(env, "java/lang/String"), jstr);
/* Run and cleanup */
(*env)->CallStaticVoidMethod(env, cls, mid, args);
(*jvm)->DestroyJavaVM(jvm);
return 0;
```
Compiling Embedded JVM and C

```java
public class JVMinCTest {
    public static void main(String[] args) {
        System.out.println("Hello from JVMinCTest");
        System.out.println("String passed from C: " + args[0]);
    }
}
```

To compile

```
$ javac JVMinCTest
$ cc JVMinC.c -o JVMinC -L /usr/lib/jvm/java-1.6.0-openjdk-1.6.0/jre/lib/i386/client/-ljvm
```

Running:

```
$ ./JVMinCTest Hello from JVMinCTest
String passed from C: from C!
$ 
```
Standard Template Library - bonus material

(Templates are examinable however, STL is NOT examinable for 2009/2010)

Alexander Stepanov, designer of the Standard Template Library says:

“STL was designed with four fundamental ideas in mind:

- Abstractness
- Efficiency
- Von Neumann computational model
- Value semantics”

It’s an example of generic programming; in other words reusable or “widely adaptable, but still efficient” code
Additional references


Advantages of generic programming

- Traditional container libraries place algorithms as member functions of classes
  - Consider, for example, "test".substring(1,2); in Java
- So if you have $m$ container types and $n$ algorithms, that's $nm$ pieces of code to write, test and document
- Also, a programmer may have to copy values between container types to execute an algorithm
- The STL does not make algorithms member functions of classes, but uses meta programming to allow programmers to link containers and algorithms in a more flexible way
- This means the library writer only has to produce $n + m$ pieces of code
- The STL, unsurprisingly, uses templates to do this
Plugging together storage and algorithms

Basic idea:

- define useful data storage components, called **containers**, to store a set of objects
- define a generic set of access methods, called **iterators**, to manipulate the values stored in containers of any type
- define a set of **algorithms** which use containers for storage, but only access data held in them through iterators

The time and space complexity of containers and algorithms is specified in the STL standard
A simple example

```cpp
#include <iostream>
#include <vector>  //vector<T> template
#include <numeric>  //required for accumulate

int main() {
    int i[] = {1,2,3,4,5};
    std::vector<int> vi(&i[0],&i[5]);

    std::vector<int>::iterator viter;

    for(viter=vi.begin(); viter < vi.end(); ++viter)
        std::cout << *viter << std::endl;

    std::cout << accumulate(vi.begin(),vi.end(),0) << std::endl;
}
```
Containers

- The STL uses containers to store collections of objects
- Each container allows the programmer to store multiple objects of the same type
- Containers differ in a variety of ways:
  - memory efficiency
  - access time to arbitrary elements
  - arbitrary insertion cost
  - append and prepend cost
  - deletion cost
  - ...
Containers

- **Container examples for storing sequences:**
  - `vector<T>`
  - `deque<T>`
  - `list<T>`

- **Container examples for storing associations:**
  - `set<Key>`
  - `multiset<Key>`
  - `map<Key,T>`
  - `multimap<Key, T>`
Using containers

```cpp
#include <string>
#include <map>
#include <iostream>

int main() {

    std::map<std::string, std::pair<int, int>> born_award;

    born_award["Perlis"] = std::pair<int, int>(1922, 1966);
    born_award["Wilkes"] = std::pair<int, int>(1913, 1967);
    born_award["Hamming"] = std::pair<int, int>(1915, 1968);
    // Turing Award winners (from Wikipedia)

    std::cout << born_award["Wilkes"].first << std::endl;

    return 0;
}
```
std::string

- Built-in arrays and the std::string hold elements and can be considered as containers in most cases
- You can’t call `.begin()` on an array however!
- Strings are designed to interact well with C char arrays
- String assignments, like containers, have value semantics:

```cpp
1 #include <iostream>
2 #include <string>

3 int main() {
4     char s[] = "A string ";
5     std::string str1 = s, str2 = str1;
6
7     str1[0]=’a’, str2[0]=’B’;
8     std::cout << s << str1 << str2 << std::endl;
9     return 0;
10 }
```
Iterators

- Containers support **iterators**, which allow access to values stored in a container
- Iterators have similar semantics to pointers
  - A compiler may represent an iterator as a pointer at run-time
- There are a number of different types of iterator
- Each container supports a subset of possible iterator operations
- Containers have a concept of a **beginning** and **end**
## Iterator types

<table>
<thead>
<tr>
<th>Iterator type</th>
<th>Supported operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>== != ++ *(read only)</td>
</tr>
<tr>
<td>Output</td>
<td>== != ++ *(write only)</td>
</tr>
<tr>
<td>Forward</td>
<td>== != ++ *</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>== != ++ * --</td>
</tr>
<tr>
<td>Random Access</td>
<td>== != ++ * -- + - += -= &lt; &gt; &lt;= &gt;=</td>
</tr>
</tbody>
</table>

- Notice that, with the exception of input and output iterators, the relationship is hierarchical.
- Whilst iterators are organised logically in a hierarchy, they do not do so formally through inheritance!
- There are also const iterators which prohibit writing to ref’d objects.
Adaptors

- An adaptor modifies the interface of another component
- For example the `reverse_iterator` modifies the behaviour of an iterator

```cpp
#include <vector>
#include <iostream>

int main() {
    int i[] = {1,3,2,2,3,5};
    std::vector<int> v(&i[0],&i[6]);

    for (std::vector<int>::reverse_iterator i = v.rbegin(); i != v.rend(); ++i)
        std::cout << *i << std::endl;

    return 0;
}
```
Generic algorithms

- Generic algorithms make use of iterators to access data in a container.
- This means an algorithm need only be written once, yet it can function on containers of many different types.
- When implementing an algorithm, the library writer tries to use the most restrictive form of iterator, where practical.
- Some algorithms (e.g. sort) cannot be written efficiently using anything other than random access iterators.
- Other algorithms (e.g. find) can be written efficiently using only input iterators.
- Lesson: use common sense when deciding what types of iterator to support.
- Lesson: if a container type doesn’t support the algorithm you want, you are probably using the wrong container type!
Algorithm example

- Algorithms usually take a `start` and `finish` iterator and assume the valid range is `start` to `finish-1`; if this isn’t true the result is undefined.

Here is an example routine `search` to find the first element of a storage container which contains the value `element`:

```cpp
// Search: similar to std::find
template<class I,class T> I search(I start, I finish, T element) {
    while (*start != element && start != finish)
        ++start;
    return start;
}
```
#include "example23.hh"

#include "example23a.cc"

int main() {
    char s[] = "The quick brown fox jumps over the lazy dog";
    std::cout << search(&s[0],&s[strlen(s)],'d') << std::endl;

    int i[] = {1,2,3,4,5};
    std::vector<int> v(&i[0],&i[5]);
    std::cout << search(v.begin(),v.end(),3)-v.begin() << std::endl;

    std::list<int> l(&i[0],&i[5]);
    std::cout << (search(l.begin(),l.end(),4)!=l.end()) << std::endl;

    return 0;
}
Heterogeneity of iterators

```cpp
#include "example24.hh"

int main() {
    char one[] = {1,2,3,4,5};
    int two[] = {0,2,4,6,8};
    std::list<int> l (&two[0],&two[5]);
    std::deque<long> d(10);

    std::merge(&one[0],&one[5],l.begin(),l.end(),d.begin());

    for(std::deque<long>::iterator i=d.begin(); i!=d.end(); ++i)
        std::cout << *i << " ";
    std::cout << std::endl;

    return 0;
}
```
Function objects

- C++ allows the function call "( )" to be overloaded
- This is useful if we want to pass functions as parameters in the STL
- More flexible than function pointers, since we can store per-instance object state inside the function
- Example:

```cpp
struct binaccum {
    int operator()(int x, int y) const { return 2*x + y; }
};
```
Higher-order functions in C++

- In ML we can write: `foldl (fn (y,x) => 2*x+y) 0 [1,1,0];`
- Or in Python: `reduce(lambda x,y: 2*x+y, [1,1,0])`
- Or in C++:

```cpp
#include <iostream>
#include <numeric>
#include <vector>
#include "example27a.cc"

int main() { //equivalent to foldl

    bool binary[] = {true,true,false};
    std::cout<< std::accumulate(&binary[0],&binary[3],0,binaccum()) << std::endl; //output: 6

    return 0;
}
```
Higher-order functions in C++

- By using reverse iterators, we can also get foldr:

```cpp
#include <iostream>
#include <numeric>
#include <vector>

#include "example27a.cc"

int main() { // equivalent to foldr
    bool binary[] = {true, true, false};
    std::vector<bool> v(&binary[0], &binary[3]);
    std::cout << std::accumulate(v.rbegin(), v.rend(), 0, binaccum());
    std::cout << std::endl; // output: 3
    return 0;
}
```