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

Generate JNI Header

Compile HelloWorld.java
$ javac HelloWorld.java

Create HelloWorld.h
$ javah HelloWorld

HelloWorldImp.c

Create a Shared Library

class HelloWorld
{
    public native void displayHelloWorld();
    static {
        System.loadLibrary("hello");
    }
    public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
    }
}

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

```java
$ java HelloWorld
Hello World!
```

Primitive Types and Native Equivalents

Some examples:
- `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.
- organised like a C++ virtual function table.

Accessing Java Strings

The jstring type is not equivalent to the C string type

```c
/* Wrong way */
JNIEXPORT jstring JNICALL Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
  jmethodID mid = (*env)->GetMethodID(env,cls,''hello'',''(I)V'');
  (*env)->CallVoidMethod(env,obj,mid,parm1);
```

Accessing Java Array

```c
/* Wrong way */
JNIEXPORT jint JNICALL Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr) {
  int i, sum = 0;
  for (i=0; i<10; i++){
    int a = (*env)->GetIntArrayElement(env,arr,i);
    sum += a;
  }
  printf("\n%ij\n",sum);
```

Calling a Java Method

1. Find the class of the object

   ```java
   Call GetObjectClass
   ```

2. Find the method ID of the object

   ```java
   CallGetMethodID, which performs a look-up for the Java method in the class.
   ```

3. Call the method

   ```java
   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)
   ```

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

```c
at java.lang.System.loadLibrary(System.java:1047)
at java.lang.Runtime.loadLibrary0(Runtime.java:840)
at java.lang.System.loadLibrary(System.java:1047)
```

```
Could not find the main class: HelloWorld. Program will exit.
Problems? set your LD_LIBRARY_PATH:
export LD_LIBRARY_PATH=
```

```
```
```c
```
```
```
```c
```
```
```c
```
```
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
```c
```
Synchronisation

- Synchronise is available as a C call
- Wait and Notify calls through JNICALL 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

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

JVMinC.c

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

int main(int argc, char *argv[]) {
  JNIEnv *env;
  JavaVM *jvm;
  JavaVMInitArgs vm_args;
  vm_args.version = JNI_VERSION_1_4;
  vm_args.options = options;
  vm_args.nOptions = 4;
  /* 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!");
  /* Run and cleanup */
  (*env)->CallStaticVoidMethod(env, cls, mid, args);
  return 0;
}
```

Compiling Embedded JVM and C

```bash
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
String passed from C: from C!
```

Additional references


Standard Template Library (STL)

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

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

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
  - ...
- The STL specifies bounds on these costs for each container type

Using containers

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

int main() {
    int i[] = {1,2,3,4,6};
    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;
    return 0;
}
```

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

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
#include <iostream>
#include <string>

int main() {
    char s[] = "A string ";
    std::string str1 = s, str2 = str1;
    str1[0] = 'a', str2[0] = 'B';
    std::cout << s << str1 << str2 << std::endl;
    return 0;
}
```

A simple example

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

int main() {
    int i[] = {1,2,3,4,6};
    std::vector<int> vi(&i[0],&i[5]);
    for(viter=vi.begin(); viter < vi.end(); ++viter) {
        std::cout << *viter << std::endl;
    }
    std::cout << accumulate(vi.begin(),vi.end(),0) << std::endl;
    return 0;
}
```

Iterator types

<table>
<thead>
<tr>
<th>Iterator type</th>
<th>Supported operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>!=</td>
</tr>
<tr>
<td>Output</td>
<td>!=</td>
</tr>
<tr>
<td>Forward</td>
<td>!=</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>!=</td>
</tr>
<tr>
<td>Random Access</td>
<td>!=</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()
{
    char one[] = "The quick brown fox jumps over the lazy dog";
    std::cout << search(&one[0],&one[strlen(one)],'d') << 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!

```cpp
template<class I, class T> I search(I start, I finish, T element) {
    while (*start != element && start != finish)
        ++start;
    return start;
}
```

### Searching over multiple containers
- Include "example23.hh"
- Include "example23a.cc"
- #include <iostream>
- #include <vector>
- main() {
    // output: 3
    std::cout << search(&two[0],&two[5],4)-two.begin() << std::endl;
    return 0;
}

### 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 T, class T element> T search(T start, T finish, T element)
{
    while (*start != element && start != finish)
        ++start;
    return start;
}
```

### Heterogeneity of iterators
- Include "example24.hh"
- #include <vector>
- Include <numeric>
- Include <iostream>
- int main()
{
    int one[] = {1,2,3,4,5};
    int two[] = {0,2,4,6,8};
    std::list<int> l(&one[0],&one[5]);
    std::deque<int> d(10);
    std::vector<int> v(&i[0],&i[5]);
    std::vector<int>::reverse_iterator i = v.rbegin();
    f or std::deque<int>::iterator i=d.begin(); i!=d.end(); ++i)
    std::cout << *i << std::endl;
    return 0;
}

We use merge to merge different element types from different containers.

### 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() {
    bool binary[] = {true, true, false};
    std::cout<< std::accumulate(binary[0],binary[3],0,binaccum())
    //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 foldl
    bool binary[] = {true, true, false};
    std::vector<bool> v(binary[0],binary[3]);
    std::cout<< std::accumulate(v.rbegin(),v.rend(),0,binaccum())
    //output: 3
    return 0;
}
```