Programmieren in C++

Prof. Dr. Christoph Stamm

Übung 1: Java Native Interface

Lernziele

- Sie repetieren den Stoff aus der Vorlesung: Struktur eines C++-Programms, einfache und strukturierte Datentypen, Arrays, Klassen, Zeigerarithmetik.
- Sie entwickeln eine einfache Matrix-Klasse in Java und C++.
- Sie integrieren Ihre C++-Matrix-Klasse in Ihren Java-Code mittels JNI.
- Sie führen Performance-Messungen aus.

1 Einführung in JNI

1.1 Eigenschaften des JNI

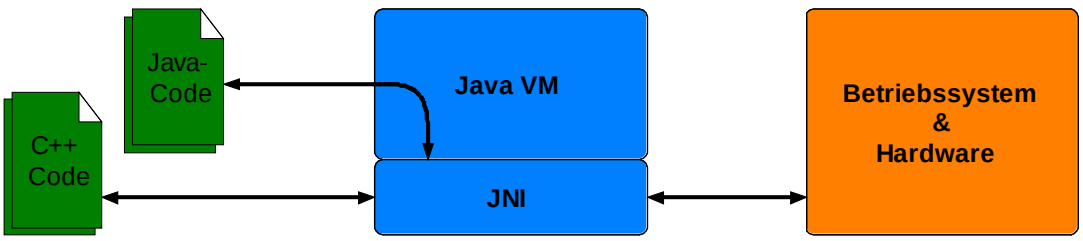
Es gibt verschiedene Gründe, weshalb es wünschenswert sein kann, aus einem Java-Programm C++-Code bzw. aus einem C++-Programm Java-Code aufrufen zu können:

- Es soll Funktionalität genutzt werden, die in der anderen Sprache implementiert ist, aber in der eigenen Sprache nicht zur Verfügung steht (zum Beispiel die Nutzung eines 3D-Toolkits).
- Obschon Java in den letzten Jahren bedeutend schneller geworden ist, reicht die Geschwindigkeit von Java-Programmen nicht immer an die eines optimal kompilierten C++-Programmes heran. Bei rechenintensiven Prozessen ist ein C++-Programm oft etwas schneller als die in Java implementierte Variante (man sollte allerdings bedenken, dass bereits der Aufruf von C++-Code aus einem Java-Programm einen erheblichen Grundaufwand erfordert. Der Gewinn muss also gross genug ausfallen, dass auch zusätzlicher Zeitverlust kompensiert werden kann).
- Die Java-VM ist mehr oder weniger plattformunabhängig und stellt deshalb gewisse Möglichkeiten einer bestimmten Plattform nicht zur Verfügung. Über ein JNI kann ein Java-Programm aber auf nativen C/C++-Code zugreifen, der plattformspezifische Funktionen zugänglich macht (so können beispielsweise Betriebssystem-Routinen nur in C++ direkt aufgerufen werden).

Das JNI bietet folgende Möglichkeiten an (als native Sprache kann bisher nur C/C++ benutzt werden):

1. Aufruf von C/C++-Funktionen, welche in einer dynamisch gebundenen Bibliothek hinterlegt sind (Windows: DLL).
2. Die aufgerufene C/C++-Funktion kann auf Java-Klassen und -Objekte zugreifen (auch Arrays, Methoden-Aufrufe).
3. Von C++ aus kann eine JVM gestartet werden, um bestehenden Java-Code nutzen zu können.

Die Architektur rund um das JNI:



Das JNI sorgt für eine standardisierte Schnittstelle zwischen einem Java- und einem C++ Programm und ermöglicht den Zugriff auf das Betriebssystem sowie die Hardware. Die Anteile von Java und C++ werden auf diese Weise miteinander verknüpft. Übrigens wird das JNI auch verwendet, wenn mit java.exe eine neue JVM gestartet wird.

JNI ist offen standardisiert, so dass Java-Code, welcher native Methoden benutzt, zwar plattformabhängig, nicht aber JVM-abhängig wird. Natürlich ist es einem JVM-Entwickler freigestellt, ob er sich an diesen Standard halten will oder nicht, aber tut er dies nicht, stellt er sich selbst ins Abseits.

1.2 Aus Java heraus nativen C/C++-Code aufrufen

Eine Methode, welche C-Code aufruft wird in Java als *native* deklariert. Sie verfügt über folgende Möglichkeiten:

- Sie kann Parameter übernehmen und einen Rückgabewert ausgeben.
- Sie kann auf Arrays, Objekte (Element-Selektion) und Klassen (statische Elemente) zugreifen.
- Sie kann auf dem Java-Heap neue Objekte erzeugen.
- Sie kann Objekte im Java-Heap fixieren, so dass sie vor Verschieben oder Löschen durch den Garbage-Collector geschützt sind.
- Sie kann neue Klassen laden, Exceptions werfen, Exceptions von Java-Methoden oder der VM fangen.

Folgende drei Schritte sind nötig, um nativen Code einer Java-Applikation zur Verfügung zu stellen:

a) Schreiben Sie ein Java-Programm, das die nativen Methoden benutzen wird bzw. erweitern Sie ein bestehendes Programm: Es sind die gewünschten nativen Methoden in Form von native Methodenköpfen zu schreiben. Ausserdem muss die zukünftig zu benutzende DLL eingebunden werden.
b) Generieren Sie mit *javac -h <output-folder> <source-file>* eine C-Header-Datei (javac -h sucht im Java Code nach nativen Methoden und erzeugt daraus JNI-kompatible Funktionsprototypen im C++ Format).
c) Eröffnen Sie ein C++-Projekt mit einer DLL als Ziel. Implementieren Sie die nativen Methoden in C++ und generieren Sie die DLL.

Im Abschnitt 2 können Sie diese drei Schritte in Form eines Tutorials für eine einfache Anwendung Schritt für Schritt durchführen und nachvollziehen.

Hintergrundinformationen

a) Die in Schritt (a) deklarierten nativen Methoden ähneln in gewisser Hinsicht den bekannten abstrakten Methoden in Java: Sie besitzen nur einen Funktionskopf und keinen Rumpf. Im Unterschied zu einer abstrakten Methode wird eine native Methode aber nicht in einer abgeleiteten Klasse, sondern in einer externen Bibliothek als C/C++-Code implementiert.
   Der erwähnte Import der Bibliothek(en) hat in einem statischen Initialisierungsblock der Klasse zu erfolgen, welche die nativen Methoden definiert.
b) Die Signaturen, der in Java definierten nativen Methoden, sind frei wählbar und müssen lediglich den üblichen Java-Konventionen entsprechen. Für die Benutzung des JNI gelten aber spezielle Regeln, die Sie aber vorerst nicht beachten müssen. Der Header-Generator übersetzt Ihre Methodennamen in gültige JNI-Namen und generiert dabei eine Header-Datei, die Sie in Ihrem C++-Projekt später benutzen werden. Diese Header-Datei besteht im Wesentlichen aus einer Ansammlung von Funktions-Signaturen gemäss JNI-Konvention.
c) In Ihrem C++-Projekt müssen Sie die im generierten Header vorgegebenen Funktionen implementieren, wobei die Funktionsnamen nicht geändert werden dürfen! Nur so findet JNI später zu jedem Aufruf einer nativen Methode die passende Funktion in der DLL. Der Header dient also als eine C++–seitige Schnittstelle zu JNI.

1.3 Datentransfer zwischen Java und C++

Wenn Sie sich die automatisch generierte Header-Datei in Kapitel 2 genau ansehen, finden Sie dort zusätzlich zu Ihrem eigenen `int`-Parameter zwei weitere, seltsam anmutende Parameter:

- Der erste Parameter `JNIEnv*` ist der JNI-Schnittstellenzeiger und ermöglicht dem C++-Code den Zugriff auf die Java VM. Über diesen Zeiger können Sie spezielle JNI-Funktionen aufrufen.
- Der zweite Parameter `jobject` bzw. `jclass` repräsentiert den Aufrufer der nativen Methode. Hierbei handelt es sich entweder um ein Objekt (`jobject`) oder im Falle eines statischen Aufrufers um eine Klasse (`jclass`). Der Aufrufer wird per Referenz übergeben.

Wir werden uns vorerst nicht weiter um die VM-Funktionen kümmern. Es sei lediglich eine kurze Übersicht der zur Verfügung stehenden Möglichkeiten aufgezeigt (weiterführende Informationen sind in der offiziellen Java-Dokumentation, Sektion JNI, Abschnitt Spezifikation, Kapitel 4 zu finden):

- Operationen auf Java-Klassen und -Objekten (z.B. Klassen laden, Objekte und Arrays erzeugen)
- Exception-Handling (werfen und geworfene empfangen)

- Globale Referenzen auf Elemente erzeugen/verwalten
- Auf Methoden und Attribute von Objekten/Klassen zugreifen
- Operationen für Arrays und Strings
- Operationen für Monitore, new I/O, Reflection

Übergabe primitiver Datentypen

Die primitiven Typen in Java sind plattformunabhängig, was für die C-Typen bekanntlich nicht gilt. Deshalb werden die Java-Typen plattformabhängig auf sogenannte primitive Systemtypen in C++ abgebildet. Dies erfolgt gemäss folgender Tabelle:

Java-Typ	C++ Typ	Speicherbedarf [Bit]	Vorzeichen
`void`	`void`	-	-
`boolean`	`jboolean`	8	nein
`char`	`jchar`	16	nein
`byte`	`jbyte`	8	ja
`short`	`jshort`	16	ja
`int`	`jint`	32	ja
`long`	`jlong`	64	ja
`float`	`jfloat`	32	-
`double`	`jdouble`	64	-

Achten Sie darauf, in Ihrer Implementation der nativen Funktionen ausschliesslich die Systemtypen zu benutzen!

Übergabe von Arrays mit einem primitiven Elementtyp

Java und C++ unterscheiden sich erheblich in der Handhabung von Arrays: Während Arrays in Java durch echte Objekt-Referenzen repräsentiert werden, wird in C/C++ für C-Arrays lediglich ein Zeiger auf einen Speicherblock benutzt. Java-Arrays, welche primitive Elemente enthalten, können durch das JNI einfach auf C-Arrays abgebildet werden, indem die entsprechenden Systemtypen von C++ benutzt werden.

Der direkte Zugriff auf ein primitives Array, wie wir von Java-Referenzen und C-Zeigern gewohnt sind, ist in einer nativen Funktion aus verschiedenen Gründen nicht möglich:

- Wegen des im Hintergrund arbeitenden Garbage-Collectors (GC) der JVM kann für einen externen Nutzer von Java-Daten nicht garantiert werden, dass sie sich stets an derselben Adresse befinden. Der GC verschiebt während seiner Aufräumarbeiten Objekte unter Umständen in andere Speicherbereiche. Nur wenn der GC es zulässt, einzelne Speicherbereiche zu fixieren, kann direkt auf den Originaldaten gearbeitet werden (aktuelle VMs unterstützen Fixierung nicht).
- Die JVM garantiert nicht, dass sich Java-Arrays in einem zusammenhängenden Speicherbereich befinden, weshalb man sich im Hinblick auf Portabilität nicht auf eine korrekte Lokalisierung eines Elements in einem Array durch die in C++ übliche Zeigerarithmetik verlassen kann.

Das JNI bildet Java-Arrays primitiver Typen auf 8 verschiedene Array-Typen in C++ ab: `jbooleanArray` bis `jdoubleArray`. Über den Umgebungszeiger `JNIEnv*` können verschiedene Operationen zum Zugriff auf Arrays durchgeführt werden. Beachten Sie, dass die Signaturen der Funktionen in der Java-Dokumentation (Kap. 4, Array-Operations) auf eine Nutzung in der C-Umgebung abgestimmt sind. Mit C++ ist der Aufruf einer JNI-Funktion einfacher und sinnreicher. Ein Beispiel:

```
// es sei env der übergebene JNIEnv-Zeiger und array eine jarray-Referenz:
jsize len = (*env)->GetArrayLength(env, array); // Aufruf gemäss Dok für C
jsize len = env->GetArrayLength(array);         // Aufruf in einer C++ Umgebung
```

Die wichtigsten JNI-Funktionen für den Array-Zugriff (für Objekt-Arrays: siehe Java-Doku) sind:

- `GetArrayLength()`
- `Get<primitiveType>ArrayElements()`
- `Release<primitiveType>ArrayElements()`
- `Get<primitiveType>ArrayRegion()`
- `Set<primitiveType>ArrayRegion()`

Für den generischen `<primitiveType>` ist der entsprechende Java-Typenname einzusetzen. Um mit einem ganzen Array arbeiten zu können, ist zunächst die Länge abzufragen. Anschliessend wird der Array in den nativen Code kopiert, wobei dafür ein Zeiger des entsprechenden Systemtyps bereitgestellt werden muss. Wenn das Array nicht mehr benötigt wird, muss es freigegeben werden (die Änderungen werden in die VM zurückkopiert).

Übergabe von Strings

Strings stellen einen Sonderfall dar, da sie erstens in Java nicht durch einen Array, sondern durch ein Objekt dargestellt werden, und zweitens Unicode-Zeichen benutzen, in C++ aber oft ASCII-Zeichen. JNI wandelt übergebene Strings entweder in das Format *modified* UTF-8 um oder speichert sie im UTF-16-Format ab. Der Zugriffsmechanismus ist ähnlich wie für Arrays. Einige wichtige Methoden sind:

- `GetStringUTFLength()`
- `GetStringUTFChars()`
- `ReleaseStringUTFChars()`

Dieselben Funktionen ohne „UTF“ verwenden 16-Bit Unicode-Zeichen. Strings werden als `jstring`-Referenzen übergeben. UTF-Konversion gibt einen `const jbyte*`, 16-Bit-Konversion an einen `const jchar*` zurück.

Übergabe von Objekten bzw. Arrays von Objekten

Im Gegensatz zu primitiven Typen und primitive Arrays gibt es keinen Weg, Java-Objekte in die C++-Umgebung abzubilden. Sie verbleiben stets im Speicherbereich der VM. Zugriffe auf Methoden und Attribute sind über entsprechende JNI-Methoden möglich. Arrays von Objekten (`jobjectArray`) können demzufolge auch nicht wie primitive Arrays in die Umgebung hereinkopiert werden, sondern müssen elementweise benutzt werden. Für die Referenzierung von Objekten stellt das JNI nur den einen Typ `jobject` zur Verfügung. Daraus lässt sich folgern, dass der Zugriff auf Objekte und Klassen wesentlich teurer als der Zugriff auf primitive Daten ist.

Beispiel: Eine Methode aufrufen (Voraussetzung: Es wurde ein `jobject`-Parameter `obj` übergeben)

1. Ermitteln der zugehörigen Klasse: `jclass c = env->GetObjectClass(obj);`
2. Abfragen des Methoden-Handles: `jmethodID m = env->GetMethodID(c, "name", "signatur");`
3. Methode (beliebig oft) aufrufen: `jType i = env->CallTypeMethod(obj, c, m, parameter);`

Die Syntax der Signaturen finden Sie in der Java-Doku, Kapitel 3, Type Signatures.

2 Eine einfache JNI-Anbindung (Tutorial)

In diesem Tutorial wird eine einfache Anbindung von C++-Code an ein Java-Programm realisiert: Die DLL soll zwei Funktionen anbieten, eine ohne und eine mit Parameter-Übergabe (primitiver Typ). Dafür müssen zwei native Methoden in Java deklariert und in einem C++-Projekt implementiert werden. Führen Sie die folgenden Schritte aus, um zum gewünschten Ergebnis zu kommen:

1. Schreiben Sie eine einfache Java-Klasse „Test“ und deklarieren Sie darin die beiden nativen Methoden

   ```
   public static native void display();
   public static native int increment(int value);
   ```

   implementieren Sie eine main-Funktion, welche die beiden nativen Methoden aufruft und fügen Sie einen statischen Initialisierer ein, der die zukünftige DLL laden wird:

   ```
   static {
      System.loadLibrary("NativeFunctions");
   }
   ```

   Kompilieren Sie die Applikation, so dass „Test.class“ erzeugt wird. Wenn Sie jetzt bereits versuchen, das Programm laufen zu lassen, bekommen Sie einen Fehler, da die DLL, welche die Implementation der beiden Funktionen hinterlegen soll, noch nicht existiert.
2. Erzeugen Sie aus „Test.java“ den Header für das C++-Projekt, indem Sie „javac -h“ aus einer Konsole heraus aufrufen, das Zielverzeichnis angeben und die java-Datei als Parameter übergeben. Die Kommandozeile in der Konsole sieht so aus:

   ```
   javac -h AusgabePfad Test.java
   ```

Als Resultat erhalten Sie „Test.h“. Der Ausgabepfad darf keine Leerzeichen enthalten oder muss in Anführungs- und Schlusszeichen gesetzt werden. Falls der Kommandozeilen-Interpreter „javac“ nicht findet, müssen Sie die Umgebungsvariable „path“ des Systems entsprechend anpassen oder den Pfad zu javac.exe angeben.

Falls Sie einmal ein etwas grösseres JNI-Projekt bearbeiten müssen, lohnt es sich, die „javac -h“-Aufrufe in eine Batch-Datei zu verpacken und direkt in das Eclipse-Projekt zu integrieren.

3. Nun sollen Sie die nativen C/C++-Funktionen implementieren und eine DLL erzeugen:

   Zunächst ist ein neues C++-Projekt für eine DLL zu eröffnen: Wählen Sie eine leere Win32-Konsolenanwendung mit einer DLL als Ziel und nennen Sie das Projekt „NativeFunctions“.

   Damit das Projekt Zugriff auf die JNI-Schnittstelle hat, müssen Sie in den Projekteinstellungen die Ihrem System entsprechenden include-Pfade hinzufügen:

   ```
   c:\Program Files\Java\jdk-xxx\include
   c:\Program Files\Java\jdk-xxx\include\win32
   ```

   Implementieren Sie die beiden nativen Methoden innerhalb von „main.cpp“ in einer sinnvollen Art und Weise, so dass `display()` einen Text auf der Konsole ausgeben soll und `increment()`, den ihr übergebenen Wert um eins erhöht zurückgibt. Die von „javac -h“ generierte Header-Datei „Test.h“ müssen Sie ihrem C++-Projekt hinzufügen und innerhalb von main.cpp includieren. Übernehmen Sie die Funktionsschnittstellen aus Ihrer Header-Datei. Diese können sich von den nachfolgenden leicht unterscheiden.

   ```
   JNIEXPORT void JNICALL Java_Test_display(JNIEnv*, jclass) {
      cout << "C++: Hello, world!" << endl;
   }
   JNIEXPORT jint JNICALL Java_Test_increment (JNIEnv*, jclass, jint i) {
      return i + 1;
   }
   ```

   Kompilieren Sie das Projekt und lassen Sie die Datei „NativeFunctions.dll“ generieren. Verschieben Sie die DLL entweder in das Java-Projektverzeichnis oder fügen Sie im Java-Projekt eine passende Umgebungsvariable hinzu, damit die DLL gefunden werden kann. Starten Sie anschliessend Ihr Java-Programm und überprüfen Sie die Funktionsweise.

3 Matrixmultiplikation (Aufgabe 1)

Nun haben Sie einen ersten Eindruck von JNI und sind in der Lage, in einem Java-Programm nativ implementierte Funktionen einzubinden und auszuführen. Diese Möglichkeit wollen wir hier nutzen, um die Matrixmultiplikation von grösseren Matrizen nach C++ auszulagern.

Die Multiplikation zweier Matrizen *A* und *B*, *R* = *A*·*B* ist wie folgt definiert: $r_{ij} = \sum_{k=1}^{v} a_{ik} b_{kj}$, wobei *A* eine u×v, *B* eine v×w und *R* eine u×w Matrix sind. Der Rechenaufwand für quadratische Matrizen der Seitenlänge *n* ist, wie aus der verwendeten Formel ersichtlich ist, relativ hoch und beträgt $O(n^3)$. Es existieren schnellere Algorithmen, wie beispielsweise der Strassen-Algorithmus, welcher mit etwa $O(n^{2.7})$ Operationen auskommt. Wir verwenden in dieser Aufgabe den langsameren Standardalgorithmus. Um für Zeitmessungen aussagekräftigere Werte zu erhalten, führen wir mehrere Multiplikationen hintereinander aus.

Auch beim Standardalgorithmus sind verschiedene Optimierungen möglich: versuchen Sie Caching-Effekte sinnvoll zu nutzen und verzichten Sie auf unnötige Multiplikationen bei der Indexberechnung.

3.1 Java-Implementierung

Implementieren Sie in Java eine Matrix-Klasse (ohne Package-Angabe), wobei eine Matrix als eindimensionales double-Array gespeichert werden soll. Die Anzahl Zeilen und Spalten wird in zwei int-Attributen festgehalten. Programmieren Sie zwei Konstruktoren: der erste hat zwei Parameter (Höhe und Breite) und initialisiert alle Matrixelemente mit Zufallszahlen im Bereich [0,1); der zweite übernimmt zusätzlich einen dritten Parameter und initialisiert alle Matrixelemente mit dem Wert des dritten Parameters. Implementieren Sie anschliessend die Methoden `Matrix multiply(Matrix m)` und `boolean equals(Matrix m)`. Die Methode `equals()` wollen wir später verwenden, um verschiedene Resultatmatrizen miteinander zu vergleichen.

3.2 C++-Implementierung

Erweitern Sie Ihre Matrix-Klasse um die Methode `Matrix multiplyNative(Matrix m)`, welche wiederum eine Matrixmultiplikation durchführt, diesmal jedoch unter Zuhilfenahme der in C++ zu implementierenden und einzubindenden Funktion:

```
native void multiplyC(double[] a, double[] b, double[] r, int m, int n, int o);
```

Achten Sie in der C++-Implementierung darauf, dass Sie mit `GetDoubleArrayElements()` einen rohen Zeiger auf das Array erhalten und dass Sie vor Beendigung der Methode mit `Release-DoubleArrayElements()` die Array-Verwendung abschliessen müssen. Beachten Sie auch die richtige Verwendung aller Parameter dieser beiden Funktionen.

3.3 Testprogramm

Schreiben Sie ein kleines Testprogramm in Java, welches eine 500×6000 mit einer 6000×400 Matrix multipliziert und die dafür notwendige Zeit ermittelt und ausgibt. Führen Sie Multiplikation einmal ganz in Java und einmal unter Einbezug der nativen C++-Funktion aus. Stellen Sie sicher, dass die beiden Resultate identisch sind. Kopieren Sie eine typische Ausgabe Ihres Programms als Kommentar ans Ende des Testprogramms.

Beachten Sie in Visual Studio, dass die Debug-Version Ihrer C++ DLL wesentlich langsamer ist als die Release-Version. Führen Sie Zeitmessungen mit beiden Versionen durch. Sie können in der Release-Version erwarten, dass die C++-Version jeweils eine Spur schneller sein wird als die Java-Version. Bei der Multiplikation wäre eine Zeit unter einer Sekunde sehr gut.

4 Matrix-Potenzierung (Aufgabe 2)

Teilen Sie zuerst Ihre Implementierung der Methode `multiply(…)` in einen öffentlichen Teil mit Resultat-Speicherallokation und Zeitmessung und einen privaten Teil mit der eigentlichen Berechnung auf. Dadurch können Sie den privaten Teil auch für die Matrix-Potenzierung verwenden.

Erweitern Sie nun Ihre Klasse Matrix um die Methode `Matrix power(int k)`, welche für ein $k > 1$ eine quadratische Matrix M potenziert, d.h. (k-1)-mal M mit sich selber multipliziert $M^k = (\ldots((M \cdot M) \cdot M) \cdot \ldots \cdot M)$. Vermeiden Sie unnötige Speicherallokationen.

Realisieren Sie danach die gleiche Funktionalität in C++.

Schliesslich ergänzen Sie das Java-Testprogramm so, dass Sie M^{91} für eine 250×250 Matrix M einmal ganz in Java und einmal unter Einbezug der C++-Funktion berechnen. Überprüfen Sie wiederum die Gleichheit der beiden Resultate und die Laufzeiten. Zur visuellen Überprüfung der Korrektheit empfiehlt es sich, eine sehr kleine Matrix zu potenzieren und auszugeben.

Auch hier können Sie erwarten, dass die C++-Version jeweils eine Spur schneller sein wird und die Rechenzeit bei ca. 1 Sekunde liegt. Wenn Sie auf unnötiges Umkopieren der Matrizen verzichten, dann schaffen Sie es sogar unter 0.15 Sekunden.