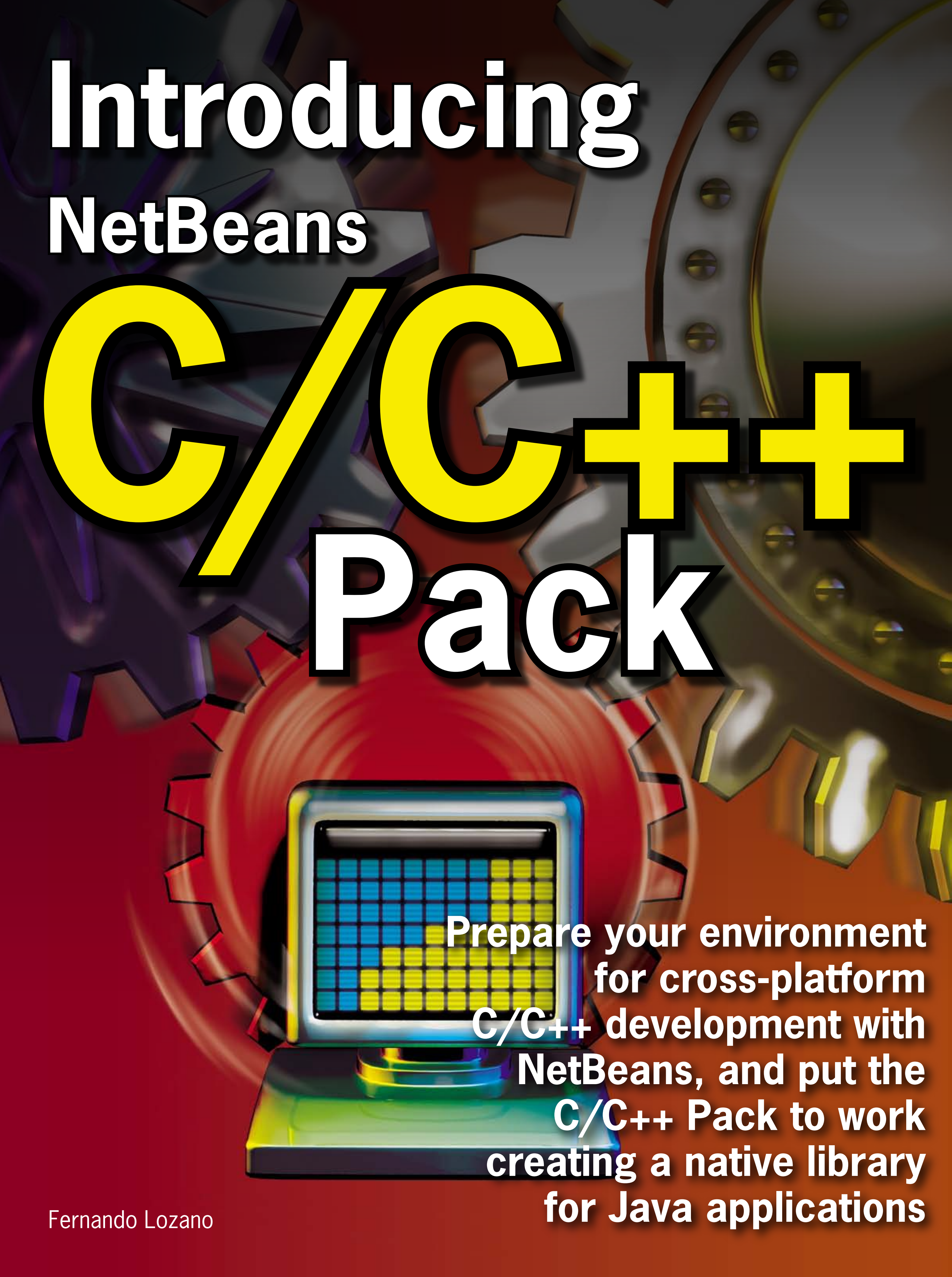
Introducing NetBeans C/C++ Pack



Prepare your environment for cross-platform C/C++ development with NetBeans, and put the C/C++ Pack to work creating a native library for Java applications

Fernando Lozano

When NetBeans 5.5 was released in late 2006, it radically changed its own value proposition by offering first-class support for a language that doesn't run inside a JVM. The NetBeans C/C++ pack provided to C/C++ programmers most features Java developers were already used to: advanced source editing with syntax highlighting and code completion, built-in CVS support, hyperlinks to navigate function declarations, a class hierarchy browser, an integrated debugger, and integration with the *make* tool.

This article focuses on how the C/C++ pack can help Java developers. Although I'm sure you all would like to code the whole world in pure Java, reality frequently challenges us to interface with native code, be it legacy systems, a device vendor SDK or a high-performance math library. Also, sometimes we need to use native code to improve the user experience, by means of tighter integration with the underlying operating system. Wouldn't it be better to do all this from the same IDE we already use for Java development?

We'll show how to leverage NetBeans and the C/C++ Pack to develop portable native libraries using C/C++, and how to integrate them with Java code in a way that eases deployment in multiple platforms.

NetBeans C/C++ Pack is more than just C/C++ coding support for the Java developer. It also suits many native code projects very well. The **sidebar** "Other open source C/C++ IDEs" compares the Pack with some popular open-source IDEs for C/C++.

Installing NetBeans C/C++ Pack

Installing the C/C++ Pack per se will be a no-brainer for most users. No matter if you've installed the NetBeans IDE using the zip package or one of the native installers, you only need to run C/C++ Pack's installer and point it to your NetBeans IDE installation directory. (Note that, although the C/C++ Pack is mostly Java code with just one tiny native library, there's no multiplatform zip archive like the ones provided for the IDE.)

The installer itself will work the same for all supported platforms: Windows, Linux and Solaris. But configuring your environment for using C/C++ Pack may not be so easy. Just like the core NetBeans IDE needs a compatible JDK installation, the C/C++ Pack will require a C/C++ compiler and standard libraries and headers. So you need to install and configure these in advance.

To meet the Pack's prerequisites, we'll rely on the popular suite formed by the GNU C Compiler (GCC), GNU Binutils, GNU Make and GNU Debugger (GDB). This is the suite that received most of the QA effort of the C/C++ Pack developer team[1], and it's portable to Windows, Linux and Solaris environments.

Using the same compiler suite for all platforms greatly simplifies dealing with portable (and even non-portable) C/C++ code, as you won't need to spend time fighting compiler directives, runtime library inconsistencies and language dialects. Besides, you'll find that in most cases the GNU toolset competes head-to-head with other C compilers in both speed and optimization quality.

Installing the GNU toolset on Linux

Linux users should have no problem obtaining the GNU toolset for their preferred platform. Mine is Fedora Core 6, and as I installed a "development workstation" using Anaconda I already had everything ready for NetBeans C/C++ Pack. Users who didn't install Linux development tools when configuring their systems should have no problem using either *yum*, *up2date*, *yast* or *apt* to install the GNU toolset.

Stay clear of CD-bootable mini-distros like Knoppix for real development work. Instead, install a full-featured distro in a native Linux partition in your main hard disk. The few additional gigabytes used will prove to be a small cost for all the hassle you'll avoid.

Solaris users will also find it easy to install the GNU toolset; there are detailed instructions on the NetBeans Web site. But be warned:

netbeans.org/products/cplusplus
NetBeans C/C++ Pack home page

[1] The only other compiler suite supported so far is the Sun Studio product for Solaris and Linux.

if you think you'd be better served by the native platform C compiler (Sun Studio), think again. This is because NetBeans C/C++ Pack's debugger *needs* the GNU Debugger, and GDB has some issues running code generated by Sun compilers. So you can use Sun's compiler to produce final code, but you'd better use the GNU toolchain for development.

Installing the GNU toolset on Windows

Windows users won't be able to use native C/C++ compilers from Microsoft, Borland or Intel, and will have to stick with a Windows port of the GNU toolset. There are two options: Cygwin and MinGW.

The C/C++ Pack's docs at *netbeans.org* provide detailed instructions for using Cygwin, but I strongly advise you to use MinGW instead. The reason is that Cygwin relies on a Unix emulation layer, while MinGW uses native Windows DLLs for everything. Code compiled with Cygwin uses the standard GNU runtime library (*glibc*) on an emulation of Unix system calls, and semantics like mount points, pipes and path separators. But code compiled with MinGW will use standard Microsoft runtime libraries such as *MSVCRT.DLL*.

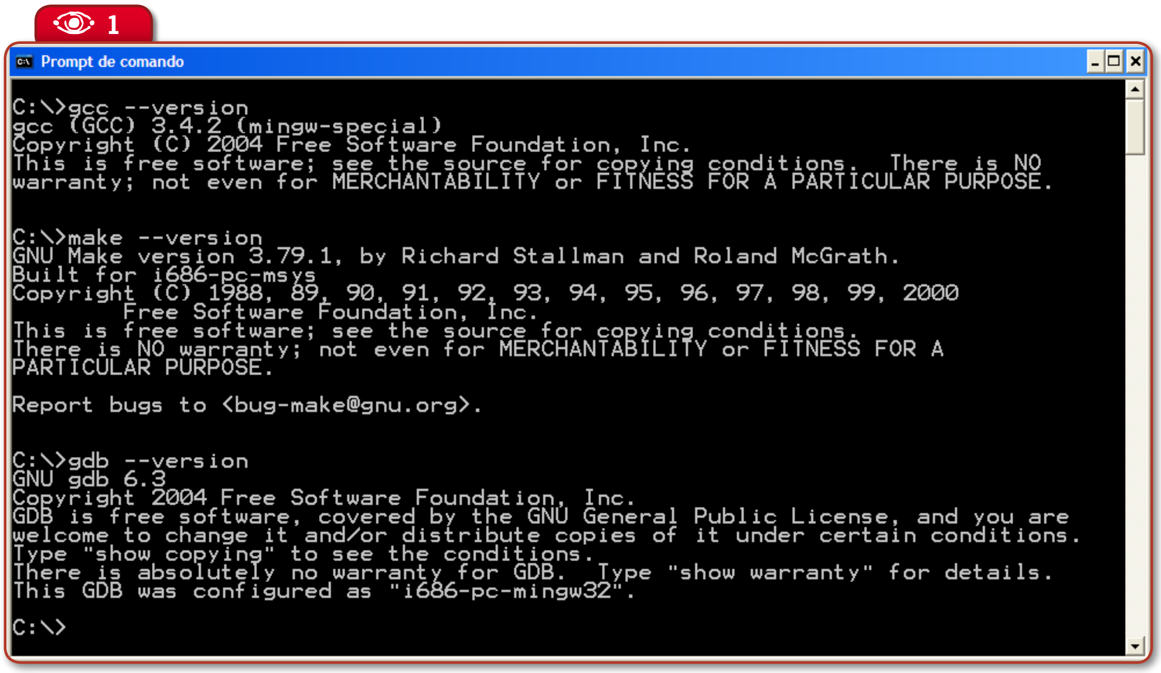
Cygwin has its uses, as many Linux and Unix software (specially open-source software) that has not yet been ported to Windows is easy to run under Cygwin without virtualization overhead. But I doubt you'd want to compromise stability and compatibility with the native platform when developing native libraries for use with Java applications. So MinGW is the way to go. The **sidebar** "Installing MinGW" provides detailed instructions.

Checking prerequisites

Whatever your platform of choice, you need access to the GNU toolset from your operating system command prompt. It may be necessary to configure the system PATH before using NetBeans C/C++ Pack. You can check that you have all prerequisites are available before proceeding by using the commands displayed in **Figure 1**. (Although this figure shows a Windows command prompt, you'll be able to run the same commands from either the Linux or Solaris shells.) If you get software releases older than the ones shown, consider upgrading your GNU toolset.

When pure Java is not enough

Now that you have NetBeans C/C++ installed and its prerequisites configured, let's present this article's use case. You're



```
C:\>gcc --version
gcc (GCC) 3.4.2 (mingw-special)
Copyright (C) 2004 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.


C:\>make --version
GNU Make version 3.79.1, by Richard Stallman and Roland McGrath.
Built for i686-pc-msys
Copyright (C) 1988, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 2000
        Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

Report bugs to <bug-make@gnu.org>.


C:\>gdb --version
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-mingw32".

C:\>
```

Figure 1 Verifying that the GNU toolset is installed and configured correctly, and is using compatible releases.

developing a desktop Java application with cryptographic features, which saves sensitive data such as key rings and private keys in a local file system folder. You want to be sure that only the user who's running the application can read (and of course write) files to that folder.

The standard Java libraries provide methods in the **java.io.File** class for checking if a file can be read or written by the current user, but these methods don't check if other users can also read or write the same files. There are new methods in Java SE 6 that deal with file permissions, and work in progress under JSR 293; but if your application has to support Java 5 or 1.4, there's no escaping from native code. So our application will use native system calls to verify local folder permissions during initialization, and refuse to start if it finds the folder is not secure.

Java doesn't provide an easy way to declare external methods, like Free Pascal or Visual Basic, but it does of course provide the Java Native Interface, a standard and portable way to call native code from Java and vice versa. With the above use case in mind, we have to design an abstraction that hides platform details and the corresponding native code from the higher application layers. In the end, the apparent complexity of dealing with JNI may actually be an advantage, because it forces us to design the interface between Java and native code, instead of just going ahead and invoking operating system APIs directly.

The Java wrapper code

Let's get our feet wet. Start NetBeans, create a Java Class Library Project, and name it "OSlib". This project will contain all interfaces between our hypothetical application and the native operating system. Then create a new class named "FilePermissions", with the code shown in **Listing 1**.

The **native** keyword, you'll remember, means that the method's implementation will be provided by a native dynamic library. That library in our code is loaded by a static initializer in the class itself.

Following Test-Driven Development practices, I'll create unit tests instead of creating a test application for the OS interface. Right click *Test Packages* in the Projects window and select *New>Test for Existing Class* to generate a skeleton for testing the native method. Then change this skeleton to make it look like **Listing 2**.

The unit tests use a properties file (shown in the same listing) to get each test's target filesystem path. This way, all file paths can be easily changed to comply with native-platform naming conventions, without needing to recompile the tests themselves. Also, don't forget to create the target files and give them appropriate permissions.

If everything is fine so far, running the tests (by selecting the **FilePermissionsTest** class and pressing *Shift+F6*) should give the output shown in **Figure 2**. The **UnsatisfiedLinkError** exception is thrown because we haven't yet provided the native method implementation.

osix.net/modules/article/?id=670

A tutorial about how to develop native GUI Windows applications using MinGW

Other open-source C/C++ IDEs

C and C++ are of course much older than Java, and are still the languages of choice for many high-profile open-source projects. Based on that, on could guess there would be many other strong cross-platform and open-source C/C++ IDEs. You'll find that NetBeans C/C++ Pack may be the strongest one around, however. Let's look at some C/C++ Pack's competitors.

DevCPP

DevCPP is very popular among Windows developers. It's lightweight, well supported, and, like NetBeans, relies on external make tools and C/C++ compilers. Additionally, it supports a wide variety of C/C++ compilers. Though DevCPP is written using Borland Delphi, an attempt to port it to Linux (using Kylix) failed. So DevCPP is not an option for cross-platform C/C++ development.

OpenWatcom

The Watcom C/C++ compiler is cross-platform but offers no Unix support; it targets Windows and OS/2. Though not very user-friendly, it comes with an integrated debugger and a help system. It was once the compiler of choice for high-performance C/C++ applications, with its enhanced code optimizer and support for all Intel processor variants. When Sybase bought Watcom, though, the C/C++ compilers and IDEs fell into obscurity. Later the tools were released as open-source software. Nowadays, it looks like the community project is going well, but there's still no support for Unix and Linux systems. This makes OpenWatcom essentially a Windows-only IDE and not suitable for our purposes.

Anjuta

Anjuta is based on the complete GNU toolset for C/C++ development. In addition to the tools supported by C/C++ Pack, it supports the GNU Autotools, a set of scripts that simplifies generating Makefiles for multiple operating systems and compilers. It's also focused on GNOME development, so it provides templates for GTK, Gnome and Glade applications.

While DevCPP and OpenWatcom are Windows-only, Anjuta and KDeveloper (see next) are Unix-only. Some users have reported success running both under Cygwin, but they are still far from providing robust support for compiling and debugging native Windows applications.

For Unix developers, Anjuta provides integrated access to *man* pages and GNOME documentation. Its integrated debugger, like C/C++ Pack, relies on GDB. The latest releases provide integration with Glade, the Gnome visual UI builder.

KDevelop

Everything said before about Anjuta applies to KDevelop, if you just replace GTK/Glade/GNOME with Qt/QtDesigner/KDE. Anjuta and KDevelop are strong C/C++ IDEs for open-source desktops, but they don't cut it as cross-platform IDEs.

Eclipse CDT

C/C++ development support in Eclipse is almost as old as Eclipse IDE itself, but it has not matured as fast as the support for Java. Although currently labeled as release 4.0, Eclipse CDT doesn't provide many features beyond those in NetBeans C/C++ Pack (which is younger).

Also like NetBeans, Eclipse CDT doesn't integrate yet with visual development tools for Gnome, KDE or Windows. It has the advantage of supporting compilers other than the GNU compilers, but this won't be a real plus if your goal is developing cross-platform C code.

Red Hat is developing GNU Autotools and RPM generation plug-ins which, when they are released as production level, may become Eclipse CDT's real advantage over NetBeans C/C++ Pack (at least for Unix/Linux users). On the other hand, NetBeans is the development IDE for Open Solaris, so don't expect it to fall short in enhancements for Unix developers.

Conclusion

The only flaw one would find in C/C++ Pack, comparing it to other open-source alternatives for C/C++ development, is the lack of operating-system and third-party library documentation support in the help system. That would be also its main drawback when compared to proprietary C/C++ IDEs. But if you evaluate alternatives for cross-platform C/C++ development, the strongest (and only) competitor for NetBeans is also its main competitor in the Java space, that is, Eclipse.

jnimaker.dev.java.net

MKO JNI Stub Maker, a NetBeans plug-in for generating JNI headers

The native code project

Our unit tests are ready, but getting native code working alongside Java code is not trivial. We'll mock the native method implementation so we can focus on how to build a native library that can be called by Java code. Start by creating a C/C++ Dynamic Library Project, as shown in **Figure 3**. Name the project "NativeOSlib" and clear the "Set as main project" checkbox.

New C/C++ projects are created empty, except for a generated *Makefile* (see **Figure 4**), and are structured in virtual folders organized by file type – not by package names like Java projects. You'll be pleased to know that NetBeans C/C++ Pack includes a Makefile editor (even though there's still no support for running arbitrary Makefile targets as there is for Ant buildfiles).

Generating JNI Stubs

We're ready to begin writing our C code. First remember that all JNI-compliant native methods should use the declaration generated by JDK's *javah* tool. You could turn to the operating system command prompt to generate the C JNI stubs, but there's a better solution. It's the JNI Maker project, a plug-in module that adds a context menu for generating JNI header files from Java classes. Just get the *nbm* package from *jnimaker.dev.java.net* and install it using NetBeans's Update Center. After restarting the IDE, you should see a new menu item as shown in **Figure 5**.

Before generating JNI stubs, make sure you've built the Java project. JNI Maker uses the distribution JARs.

Now select *Generate JNI Stub* from the **FilePermissions** class's context menu. NetBeans shows a standard *File Save* dialog, where you can select a folder to save the generated *FilePermissions.h* header file. Move into the *NativeOSlib* project folder and create a new *src* folder (C/C++ Projects do not have a default file structure with separate source and test folders like Java projects do). Save the header file there. The output window will look like **Figure 6** if the operation is successful.

JNI Maker Release 1.0 will only work correctly under Windows, but the generated code will compile and run fine on Unix/Linux. The project developers have been contacted about the module's cross-platform issues and by the time you read this there should be a new release that will work on all platforms supported by NetBeans C/C++ Pack

Listing 1. FilePermissions.java – Utility class with a native method.

```
package org.netbeans.nbmag3.util;

import java.io.File;

public class FilePermissions {

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

  public FilePermissions() {}

  // Checks if a file or folder can only
  // be read/written by the current user
  public static native boolean isPrivate(String path);
}
```

Using the JNI Maker module has the same effect as running the following command from the operating system prompt, assuming the *OSlib* project folder is the current directory and *NativeOSlib* project folder is a sibling:

```
$ javah -classpath dist/OSlib.jar -jni -o ../NativeOSlib/src/
FilePermissions.h
    org.netbeans.nbmag3.util.FilePermissions
```

java.sun.com/javase/6/docs/technotes/guides/jni
Java Native Interface specification and related tips

mingw.sf.net
MinGW, the native GNU toolset for Windows

Listing 2. Unit tests for FilePermissions native methods

FilePermissionsTest.java

```
package org.netbeans.nbmag3.util;

import java.util.Properties;
import junit.framework.*;
import java.io.File;

public class FilePermissionsTest extends TestCase {
  Properties paths = null;

  public FilePermissionsTest(String testName) {
    super(testName);
  }

  protected void setUp() throws Exception {
    paths = new Properties();
    paths.load(this.getClass().getResourceAsStream(
       "/paths.properties"));
  }

  protected void tearDown() throws Exception {}

  public void testIsPrivateOk() {
    String fileName = paths.getProperty(
        "FilePermissions.test.privateOk");

    assertTrue("File does not exist",
        new File(fileName).exists());

    boolean result = FilePermissions.isPrivate(fileName);
    assertEquals(true, result);
  }

  public void testCanReadButNotWrite() {
    boolean result = FilePermissions.isPrivate(
        paths.getProperty(
            "FilePermissions.test.readButNotWrite"));
    assertEquals(false, result);
  }

  public void testCanBeReadByOthers() {
    boolean result = FilePermissions.isPrivate(
        paths.getProperty(
            "FilePermissions.test.readByOthers"));
    assertEquals(false, result);
  }

  public void testCanBeWrittenByOthers() {
    boolean result = FilePermissions.isPrivate(
        paths.getProperty(
            "FilePermissions.test.writtenByOthers"));
    assertEquals(false, result);
  }
}
```

paths.properties

```
# For testing under Linux / Unix
FilePermissions.test.privateOk =
   /home/fernando/privateOk
FilePermissions.test.readButNotWrite =
   /home/fernando/readButNotWrite
FilePermissions.test.readByOthers =
   /home/fernando/readByOthers
FilePermissions.test.writtenByOthers =
   /home/fernando/writtenByOthers

# For testing under Windows
#FilePermissions.test.privateOk = C:\\test\\privateOk
#FilePermissions.test.readButNotWrite =
   C:\\test\\readButNotWrite
#FilePermissions.test.readByOthers =
   C:\\test\\readByOthers
#FilePermissions.test.writtenByOthers =
   C:\\test\\writtenByOthers
```

Other open-source C/C++ IDEs

bloodshed.net/devcpp.html, openwatcom.org, anjuta.sf.net, kdevelop.org

SourceForge file releases for MinGW

sf.net/project/showfiles.php?group_id=2435

Cygwin, a GNU toolset for Windows.

cygwin.com

Installing MinGW

The MinGW project provides a native port of the GNU toolset for Windows platforms. Included in the base distribution are GNU C, C++, Objective-C, Ada, Java and Fortran compilers, plus an assembler and a linker; there's also support for dynamic libraries and Windows resource files. Additional packages provide useful tools like Red Hat Source Navigator, Insight GUI debugger and a handful of Unix ports like the *wget* download manager.

MinGW stands for "Minimalist GNU for Windows". But it's "minimalist" only when compared to the Cygwin environment. (Cygwin tries to emulate a full Unix shell, complete with bash scripting, user commands and a Unix-like view of the filesystem.)

In fact, MinGW is complete to the point of providing Win32 API header files, and many popular open-source applications like Firefox have their Windows releases compiled using it. (Recent Cygwin releases include many MinGW enhancements as a cross-compiling feature, showing how Windows development is "alien" to MinGW alternatives.)

If you check the project's website, it looks like MinGW development has been stalled for quite a few years; the problem is that the site was automatically generated by a script that read the project's SourceForge area, and developers simply got tired of catching up with *sf.net's* design changes. However, MinGW is a very healthy project with active mailing lists and frequent file releases.

There is an installer for the base distribution named *mingw-x.x.exe* that downloads selected packages from SourceForge and installs them. The same installer can be used to update an existing MinGW installation.

Individual packages are downloaded to the same folder where the installer was started. This allows you to later copy the entire folder to another workstation and install MinGW there without the need of an Internet connection. Most extra packages provide their own installers or can simply be unpacked over an existing MinGW installation.

To satisfy C/C++ Pack's prerequisites, you'll need to download and install three MinGW packages: the base distribution itself, the GDB debugger, and the MSys distribution.

Installing MinGW

Download *MinGW-5.1.3.exe* (or newer) from the project's current file releases at *sf.net/project/showfiles.php?group_id=2435*, then launch it to see a standard Windows installer.

On the third step of the wizard (the second screen in **Figure S1**) you only need to select "MinGW base tools" and optionally "g++ compiler". Also, the Java Compiler may be interesting to play with, because of its ability to generate native machine code from Java sources and bytecodes, but it's not supported by NetBeans yet. Interestingly, the g77 (Fortran) compiler will be officially supported very soon.

After downloading all selected packages, the installer will ask for the destination directory and unpack all packages there. It's left to the user to configure environment variables so that MinGW tools can be used from the Windows command prompt.

Installing GDB

As we've seen, NetBeans C/C++ Pack needs GDB to be able to debug C programs. The MinGW distribution packages GDB as a stand-alone installer.

At the time of writing, the latest stable MinGW package for GDB was release 5.2.1, which won't refresh the NetBeans debugger's Local Variables window correctly. To solve this, download *gdb-6.3-2.exe* (or newer) from MinGW Snapshot Releases to a temporary folder and run it. Though you don't need to install GDB over MinGW, your life will be easier if you do, as you won't need to add another folder to your PATH system environment variable.

Installing MSys

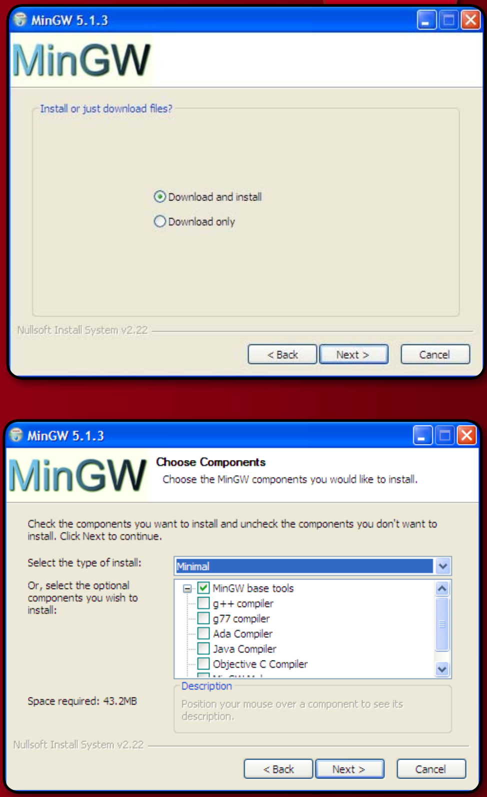
The MinGW base distribution already includes a make tool named *mingw32-make.exe*, but NetBeans C/C++ Pack won't be happy with it. MinGW's make tool is patched to be more compatible with other native Windows C compilers, and NetBeans expects a Unix-style make tool. NetBeans generated Makefiles even expect to find standard Unix file utilities such as *cp* and *rm*.

The MinGW MSys package satisfies these dependencies. It is a "Minimal System" that provides a Unix-style shell and file utilities, and allows open-source projects based on GNU Autotools to be easily built using MinGW.

Download *msys-1.0.10.exe* or newer to a temporary folder and start it. At the final installation step, a batch script configures the integration between MSys and MinGW. You will still have to add the MSys programs folder to the system PATH (in my case, *E:\MSys\1.0\bin*), as you did for the MinGW base distribution.

That's it. After running three installers and downloading about 23 MB, we are ready to develop C/C++ applications and libraries using the NetBeans IDE and C/C++ Pack on Windows.

S1



( ) Download and install [✓]
( ) Download only

Nullsoft Install System v2.22

Choose Components
Choose the MinGW components you would like to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select the type of install: Minimal

Or, select the optional components you wish to install:

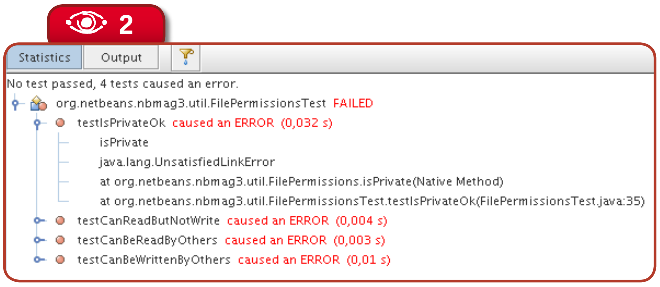
- [✓] MinGW base tools
- [ ] g++ compiler
- [ ] g77 compiler
- [ ] Ada Compiler
- [ ] Java Compiler
- [ ] Objective C Compiler

Space required: 43.2MB

Description
Position your mouse over a component to see its description.

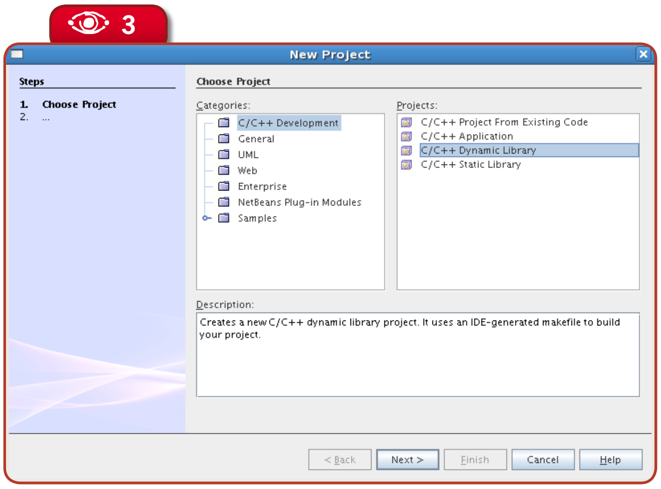
Nullsoft Install System v2.22

Figure S1 Screens from MinGW's base distribution installer.



```
No test passed, 4 tests caused an error.
org.netbeans.nbmag3.util.FilePermissionsTest  FAILED
  testIsPrivateOk  caused an ERROR  (0,032 s)
    isPrivate
    java.lang.UnsatisfiedLinkError
    at org.netbeans.nbmag3.util.FilePermissions.isPrivate(Native Method)
    at org.netbeans.nbmag3.util.FilePermissionsTest.testIsPrivateOk(FilePermissionsTest.java:35)
  testCanReadButNotWrite  caused an ERROR  (0,004 s)
  testCanBeReadByOthers  caused an ERROR  (0,003 s)
  testCanBeWrittenByOthers  caused an ERROR  (0,01 s)
```

Figure 2 Running JUnit tests for the unfinished native method.



Creates a new C/C++ dynamic library project. It uses an IDE-generated makefile to build your project.

Figure 3 Creating a C/C++ Project in NetBeans.

(The command is broken to fit the column width, but it should be typed in a single line, of course.)

Now add the generated C header file to the NativeOSlib project. Right click *Header Files* inside the *NativeOSlib* project folder in NetBeans' Projects window, and select *Add Existing Item*. Then browse to the file *src/FilePermissions.h* and open it. The code will look like **Listing 3**.

Copy the C stub function declaration from *FilePermissions.h* to *FilePermissions.c* and change it to include the header file. Also add parameter names. The code should look like **Listing 4**. (**Listing 3** highlights the declaration you have to copy, and **Listing 4** highlights the changes after copying.)

At this point, Unix and Linux users should be ready to build the native code and run unit tests again[2]. But Windows users first have to change a few project properties to make MinGW generate Windows-compatible JNI DLLs. The **sidebar** "JNI and MinGW" details these configurations.

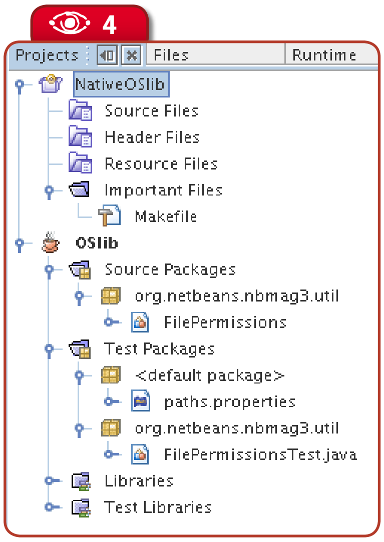
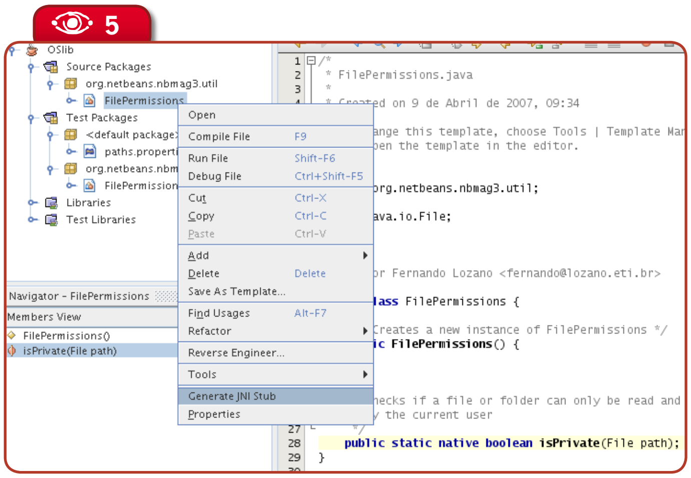


Figure 4 The new C/C++ project in NetBeans' Projects window.

Mocking native code

Due to space constraints, we won't show you the final C code for the **FilePermissions.isPrivate()** native method, but the sources available for download will provide working implementations for both Windows and Unix (Posix) systems.

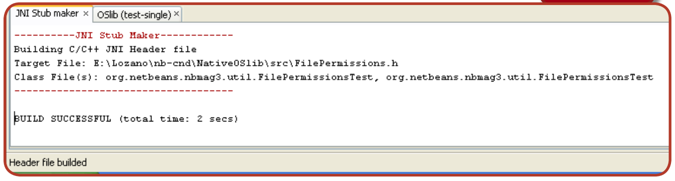
To create the C implementation file, right click *Source Files* and select *New>Empty C File*, then type "FilePermissions.c" as the file name and "src" as the folder name. A new node named *FilePermissions.c* should be created under *Source Files*.



```
/*
 * FilePermissions.java
 *
 * Created on 9 de Abril de 2007, 09:34
 ange this template, choose Tools | Template Mar
 pen the template in the editor.
 rg.netbeans.nbmag3.util;
 ava.io.File;
 r Fernando Lozano <fernando@lozano.eti.br>
 lass FilePermissions {
 Creates a new instance of FilePermissions */
 ic FilePermissions() {
 ecks if a file or folder can only be read and
 the current user
     */
    public static native boolean isPrivate(File path);
}
```

Figure 5 Generating a JNI stub using the JNI Maker plug-in module.

6



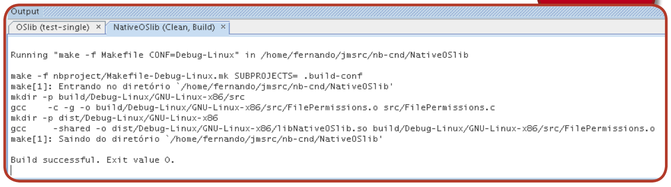
```
----------JNI Stub Maker------------
Building C/C++ JNI Header file
Target File: E:\Lozano\nb-cnd\NativeOSlib\src\FilePermissions.h
Class File(s): org.netbeans.nbmag3.util.FilePermissionsTest, org.netbeans.nbmag3.util.FilePermissionsTest
------------------------------------

BUILD SUCCESSFUL (total time: 2 secs)
```

Header file builded

Figure 6 Output from the Generate JNI Stub command.

7



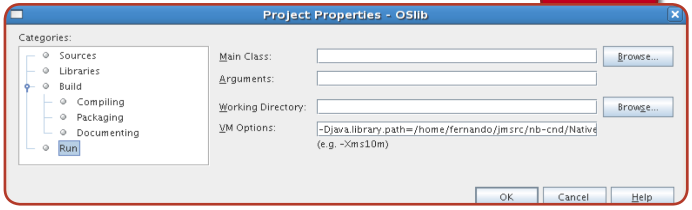
```
Running "make -f Makefile CONF=Debug-Linux" in /home/fernando/jmsrc/nb-cnd/NativeOSlib

make -f nbproject/Makefile-Debug-Linux.mk SUBPROJECTS= .build-conf
make[1]: Entrando no diretório `/home/fernando/jmsrc/nb-cnd/NativeOSlib'
mkdir -p build/Debug-Linux/GNU-Linux-x86/src
gcc    -c -g -o build/Debug-Linux/GNU-Linux-x86/src/FilePermissions.o src/FilePermissions.c
mkdir -p dist/Debug-Linux/GNU-Linux-x86
gcc     -shared -o dist/Debug-Linux/GNU-Linux-x86/libNativeOSlib.so build/Debug-Linux/GNU-Linux-x86/src/FilePermissions.o
make[1]: Saindo do diretório `/home/fernando/jmsrc/nb-cnd/NativeOSlib'

Build successful. Exit value 0.
```

Figure 7 Building the NativeOSlib project under Linux.

8



Main Class:
Arguments:
Working Directory:
VM Options: -Djava.library.path=/home/fernando/jmsrc/nb-cnd/Native
(e.g. -Xms10m)

Figure 8 Configuring the *java.library.path* property so unit tests can find the native code library on Linux.

Right click the *NativeOSlib* project and select *Clean and Build Project*. If there are no errors, you should see *make*'s output as in **Figure 7**.

Running unit tests again

You need to set the *OSlib* project's **java.library.path** system property before running it, or you'll still get **UnsatisfiedLinkError** exceptions. Open the project's Properties dialog, select the *Run* category and change *VM Options* to specify the full path to the *NativeOSlib* project's platform-specific native-library folder, which is inside the *dist* folder (see **Figure 8**). In Linux, this will be *PROJECT_HOME/dist/Debug/GNU-Linux-x86*; in Windows, *PROJECT_HOME\dist\Debug\GNU-Windows*.

Now run the unit tests again. The result should be as shown in **Figure 9**. Since the mock native code always returns true, some tests pass even if you have not created target test folders or forgotten to setup their access permissions. Anyway, the first test should fail because it takes an extra step to check if the target file path actually exists.

Managing platform-specific compiler settings

NetBeans C/C++ Pack puts object files in the *build* and *dist* folders, inside subdirectories named after the target platform, for example *GNU-Linux-x86* or *GNU-Windows*. But it won't save different compiler options for each target, forcing you to have a different project for each platform if there's a need for platform-specific compiler settings.

Listing 3. FilePermissions.h – JNI Stub for native methods in the FilePermissions class.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class
   org_netbeans_nbmag3_util_FilePermissions */

#ifndef _Included_org_netbeans_nbmag3_util_FilePermissions
#define _Included_org_netbeans_nbmag3_util_FilePermissions
#ifdef __cplusplus
extern "C" {
#endif

/*
 * Class:     org_netbeans_nbmag3_util_FilePermissions
 * Method:    isPrivate
 * Signature: (Ljava/io/File;)Z
 */
JNIEXPORT jboolean JNICALL
Java_org_netbeans_nbmag3_util_FilePermissions_isPrivate(
   JNIEnv *, jclass, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

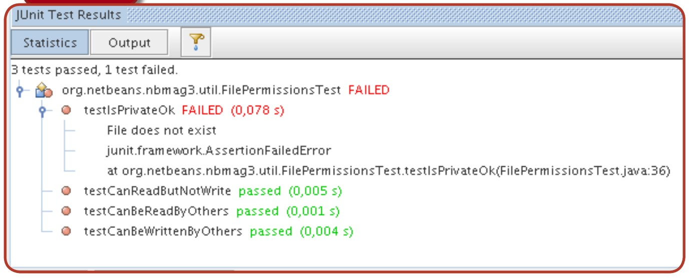
Listing 4. FilePermissions.h – JNI mock implementation for the FilePermissions native methods.

```
#include "FilePermissions.h"

JNIEXPORT jboolean JNICALL
Java_org_netbeans_nbmag3_util_FilePermissions_isPrivate(
   JNIEnv *env, jclass clazz, jstring path)
{
  return JNI_TRUE;
}
```

[2] At least if you use JDK packages compatible with your distro package manager, like the IBM and BEA JDKs provided by RHEL and SuSE Enterprise, or the RPM Packages from jpackage.org. If not, you'll have to add your JDK include folder to the GNU C compiler include directory. The configurations will be similar to the ones presented in the "JNI and MinGW" sidebar, but you won't need to change either the linker output file name or additional compiler options.

9



```
JUnit Test Results
Statistics   Output
3 tests passed, 1 test failed.
org.netbeans.nbmag3.util.FilePermissionsTest  FAILED
    testIsPrivateOk  FAILED  (0,078 s)
        File does not exist
        junit.framework.AssertionFailedError
        at org.netbeans.nbmag3.util.FilePermissionsTest.testIsPrivateOk(FilePermissionsTest.java:36)
    testCanReadButNotWrite  passed  (0,005 s)
    testCanBeReadByOthers  passed  (0,001 s)
    testCanBeWrittenByOthers  passed  (0,004 s)
```

Figure 9 Running unit tests using a mock native implementation.

You can solve this using NetBeans C/C++ Pack's multiple configurations feature. Open *NativeOSlib*'s project properties and notice the *Configuration* combo box on the top of the window (**Figure 10**). The default configurations are meant to save different compiler settings for Debug and Release

JNI and MinGW

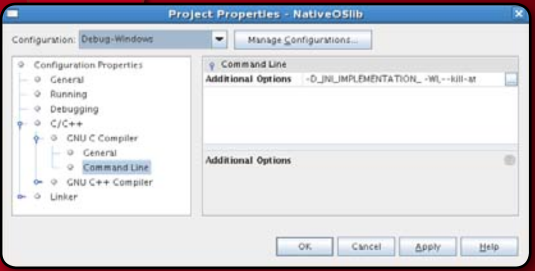
Unix and Windows native C/C++ compilers use different conventions for mangling function names[1*], exporting global symbols from libraries and setting up stack frames. JNI on Windows uses Microsoft conventions for Windows DLLs, while GCC uses its own conventions for dynamic libraries. This means that if you simply try to compile and link a dynamic library, MinGW will stick to its Unix origins and produce a DLL that is incompatible with native Windows C/C++ compilers. The JVM won't be able to get native method implementations from that library and will generate more **UnsatisfiedLinkException**s.

The solution is to add a few command-line options when compiling C/C++ sources: **-D_JNI_IMPLEMENTATION -Wl,--kill-at**. Open the C/C++ Dynamic Library Project properties and expand *C/C++>Command Line*, then type these options in the *Additional Options* text field (see **Figure S1**).

You also need to add your JDK include folders (*JAVA_HOME\include* and *JAVA_HOME\include\win32*) to the project properties. Open *C/C++>GNU C Compiler>General* and change the *Include Directories* field as shown in **Figure S2**.

You need one last change in the C/C++ Dynamic Library Project properties so you can generate a JNI-compatible DLL. By default, NetBeans chooses a library name that corresponds to Cygwin conventions, but we need to use native Windows conventions. So you need to enter the *Linker>General* category and remove the "cyg" prefix from the *Output* field (**Figure S3**).

S1

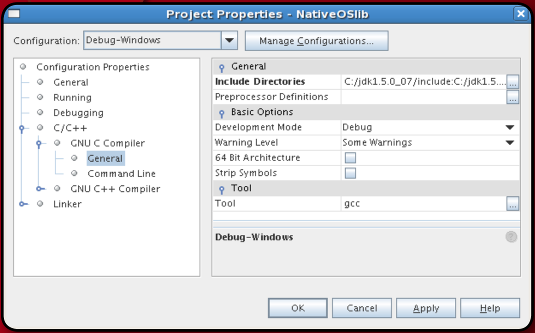


Command Line
Additional Options: -D_JNI_IMPLEMENTATION_ -Wl,--kill-at

Additional Options

Figure S1 MinGW compiler options for generating JNI-compatible DLLs

S2

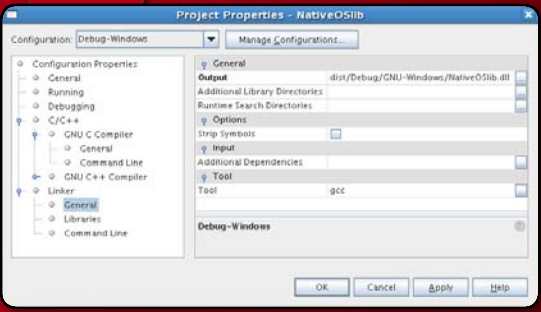


General	
Include Directories	C:/jdk1.5.0_07/include:C:/jdk1.5....
Preprocessor Definitions	
Basic Options	
Development Mode	Debug
Warning Level	Some Warnings
64 Bit Architecture	☐
Strip Symbols	☐
Tool	
Tool	gcc

Debug-Windows

Figure S2 Configuring JDK include folders for MinGW

S3

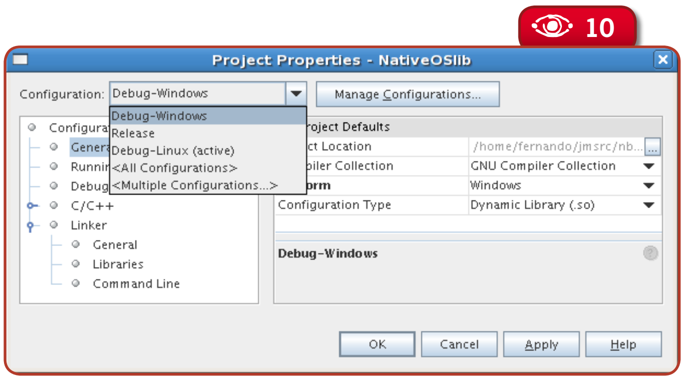


General	
Output	dist/Debug/GNU-Windows/NativeOSlib.dll
Additional Library Directories	
Runtime Search Directories	
Options	
Strip Symbols	☐
Input	
Additional Dependencies	
Tool	
Tool	gcc

Debug-Windows

Figure S3 Changing the output file name for compliance with Windows DLL naming conventions

[1]"Mangling" is the process used for generating public C++ function names in object files. It's needed because the C language doesn't support function overloading, and, to keep backward compatibility, C++ compilers generate a function name that encodes parameter types.



Configuration: Debug-Windows

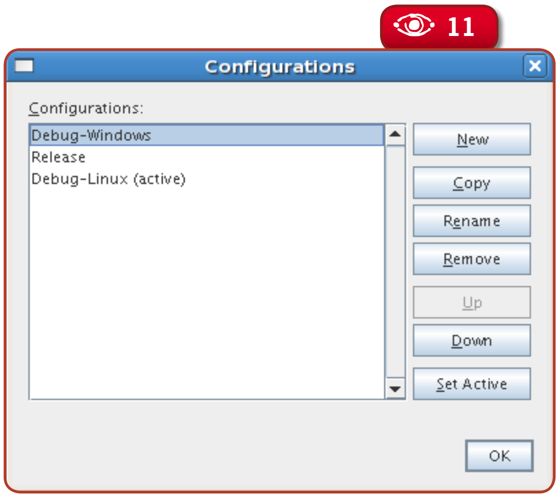
Debug-Windows
Release
Debug-Linux (active)
<All Configurations>
<Multiple Configurations...>

Project Defaults

ct Location	/home/fernando/jmsrc/nb...
piler Collection	GNU Compiler Collection
orm	Windows
Configuration Type	Dynamic Library (.so)

Debug-Windows

Figure 10
Combo box for changing compiler configurations for a C/C++ project.



Debug-Windows
Release
Debug-Linux (active)

Figure 11
Creating, renaming or copying configurations.

builds, like keeping symbol information for Debug builds and optimizing code for Release builds. So if you want platform-specific configurations, you may need to create Release and Debug variants for each platform.

The *Manage Configurations* button to the side of the combo box lets you create new configurations either from scratch or as a copy of an existing configuration (see **Figure 11**). You'll notice I renamed the generated Debug configuration to *Debug-Linux* and copied it to a new configuration named *Debug-Windows*. Doing this lets you change the Windows configuration to include all options needed by MinGW for generating JNI-compatible DLLs, while keeping the default settings for the Linux configuration.

NetBeans-generated Makefiles provide many extension points (like the Ant build-files generated by the IDE), and they can be used outside the IDE. For example, for building the *Debug-Windows* configuration you'd type the following command at the operating system prompt:

```
make CONF=Debug-Windows
```

Thus, you could have Continuous Integration servers for many platforms, all being fed by the same CVS or Subversion source tree. And thanks to GNU C cross-compiler features it would be possible to have a "compile farm" that generates native binaries for multiple platforms, without the need for multiple OS installations. For example, a Linux server could generate both Windows and Solaris SPARC binaries.

Conclusions

NetBeans C/C++ Pack provides a rich environment for developing C and C++ applications and libraries. It's useful for Java developers that need to interface with native code and, of course, for developing fully-native applications. Compiler configuration may pose some challenges for Windows developers if they never tried GNU compilers before, but the effort will certainly pay off because of the increased portability of both code and Makefiles.

Fernando Lozano (*fernando@lozano.eti.br*) is an independent consultant and has worked with information systems since 1991. He's the Community Leader of the Linux Community at Java.net, webmaster for the Free Software Foundation and counselor to the Linux Professional Institute. Lozano helps many open-source projects and teaches at undergraduate and postgraduate college courses. He's also a technical writer and book author, as well as Contributing Editor at Java Magazine (Brazil) and freelance writer for other leading IT publications.