Lab: Eclipse and C++

Steven J Zeil

February 13, 2013

Contents

An *Integrated Development Environment* (IDE) is a software package that attempts to combine the primary programming activities of project setup, editing, building (compiling), and debugging into one unified user interface. Some IDEs go even further, offering support for testing, version control, team collaboration and other activities that are beyond the scope of this course.

Any IDE will provide a facility for editing your code, compiling it, and correcting errors. Most require you to start by defining a "project", a kind of organizing unit that keeps track of which files are in your project and of any special settings or options you might need when building (compiling) your project.

Simple IDEs often assume that the only steps involved in building a project is to compile the source code. More sophisticated ones may allow you to specify certain other steps that might be required, such as constructing data files that the program will require when run, or packaging up the program and its data files into an archive file for easy delivery to others, or even running other programs that generate parts of the source code for your program.

Beginning programmers might not need all that flexibility, but the best IDEs keep all that out of your way until you actually need it.

Eclipse is a multi-platform IDE that is equally at home on both Windows and *nix systems. Eclipse may be the most popular IDE for use with Java programming. It has a flexible plug-in system that allows additions of support for many different programming tools that “real” programmers use “in the field”, including support for C++ programming.

1 Eclipse

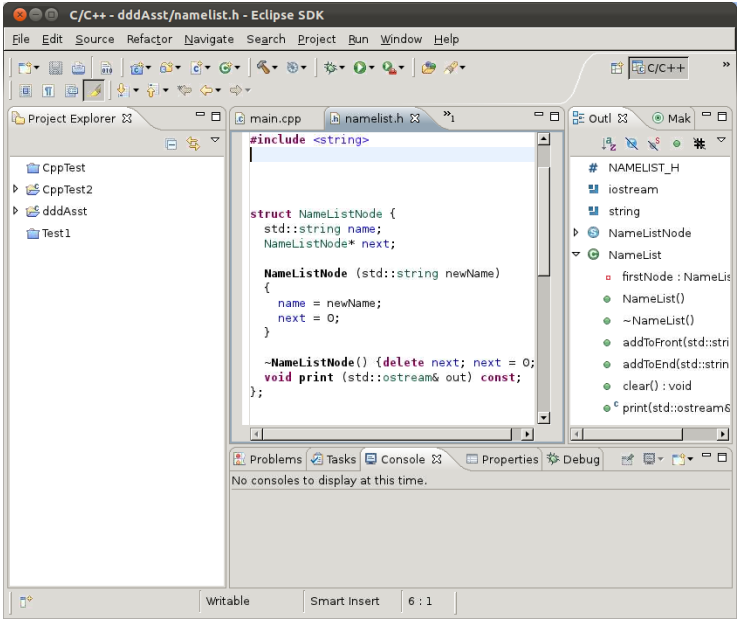
In this section we will focus on Eclipse’ support for project setup, editing, and compilation.

Eclipse is installed on our CS Dept. lab Windows PCs and on the virtual PC lab. If can also be invoked on our Linux servers via the command

```
eclipse &
```

Eventually you may want to install Eclipse on your own PC and/or in your own account area on the Cs Dept machines (so that you can play with different plugins than have been installed for general use).

One typically begins work with Eclipse by creating a project. By default, Eclipse gathers all of your projects together under a common directory called your *workspace*. Each project occupies its own subdirectory within the workspace. However, if you prefer, you can override this default and keep your projects wherever you like. If you already have some source code files, you would probably set up the project where these reside. You can then edit existing files or create new ones.



Eclipse can complete partially-typed names and offer suggestions as to what may legally be placed at the cursor position. Type Ctrl-space to see this help. It also may offer such help if you simply pause for a moment at a suggestive spot (e.g., immediately after a '.').

The Eclipse editor has another feature that sets it above moe IDEs. While you are typing your code, it is checking continuously for errors. Almost all Java compilation errors and many C++ errors will actually be flagged for you as you commit them, instead of waiting until you explicitly request a compilation/build of your project.

When you are ready, you can attempt a project build, via the menus or the toolbar buttons (hover your mouse over each button to discover what it does. If you have error messages, selecting one will take you to the relevant code location.

Once your errors have been dealt with, you can run the project or launch the debugger. You can do this via the menus, the toolbar buttons, or by right-clicking on the executable file in the Project Explorer and selecting the relevant action.

Eclipse has some truly interesting features, especially for Java. For example, it "knows" about many of the common transformations that programmers make to their code and can apply them in one step. These are called *refactoring*s. A basic example of refactoring is renaming a variable or function - Eclipse not only changes the name where the variable/function is declared, but also all the places where it is used, and it is smart enough to distinguish between uses of different functions or variables that happen to have the same name. Another refactoring is to "encapsulate" a data member, making the data member private and creating get/set functions for public access to it, all in one step.

Eclipse also understands the basic structure of the language. If you select a variable or function name with the mouse, you can ask Eclipse to take you to the line of code where that name is declared. Or you can select a declaration and ask Eclipse to list all the places in your code where that name is used.

2 Editing & Compiling

If you have already done the Eclipse Java lab in Windows, I recommend that you do this lab in Linux, or if you did the Java lab in Linux, try this one in Windows.

If you are working on a remote Linux machine, remember that you will need to be running X (or, especially from off-campus, NX).

1. Get the starting source code for this lab here. Unpack into any convenient directory.

2. Now, run Eclipse.

   You will be asked where you want to keep your workspace (the default area for all of your project settings). Accept the suggestion or browse to a more convenient directory.

   If this is your first time running Eclipse, you will be taken to a Welcome page. Close that or click on the curved arrow on the right to enter your workbench.

3. Select "`New->Project...`" from the File menu or by right-clicking in the Package explorer pane on the left. Select "C++ Project" and click Next.

   Give your project a conventient name (e.g., “findPrimes”), clear the "Use default location" box, and for "Folder to create project in:", browse to the directory where you unpacked the source code. Accept the rest of the defaults, clicking Next until you come to Finish, then click that.

   A box will come up asking if you want to open the C++ perspective. Click in the "Remember my decision box" and then click "Yes". (You probably want to do this whenever you are asked about opening a new perspective).

4. If you expand the listing, in your Project Explorer pane, you will see all of the files and directories in your project directory. Eclipse has added a new directory, the Debug directory, where it will store your compiled binaries. Eclipse should have

already found your .cpp and .h files. You won't need to add them to the project. In fact, Eclipse may have already compiled them. You can expand the Errors item in the Problems pane to see the resulting messages. Double-click on any message and the editor will show you he relevant code location.

Double-click on `findPrimes.cpp` to open it in the editor (if it's not already open).

Click on the lower of the two red rectangles on the right of the listing just look for the line near the bottom containing a call to a function named "find". Hover your mouse pointer over the line, and the text of the error message will pop up. Position your cursor just after the "d" in "find". Now pretend that you had just been typing "find", then realized that you weren't sure what the actual name of the function was that you wanted. Type Ctrl-space and a suggestion box will pop up. Three functions are suggested: the "find" from <iostream> and the functions findPrimes and findSomething, both of which are functions in this program.

Double-click on findPrimes to select it. Eclipse inserts the rest of the function name. Unfortunately, it also adds a bogus set of parentheses "()" that we don't need. Delete these.

5. Now move down to the next line and delete the semicolon. Notice that an error marker pops up almost immediately on the left. Eclipse will actually find many C++ mistakes as you type them, before you actually run the compiler. Restore the semicolon and save the file.

6. Now try to build the project via the Project menu, or via the Build item in the toolbar (usually a hammer). You should see a shorter list of compilation errors in the "Problems" pane, and the red marker next to the "find..." line should go away. Double-click on the first error message, which will take to a line inside sieve.h, and change "integer" to "int". Save the file, and build again.

This time you should succeed with no errors.

7. Run the program via the Run toolbar button (usually a white triangle in a green circle) or via the Run menu. You will see the prompt appear in the Console pane. Enter a moderately-sized integer (say something in the 3 or 4-digit range). You should see a list of prime numbers go scrolling by. Hit Enter to close that window.

8. Outside of Eclipse, examine the contents of your project directory. You can find your compiled executable in Debug/.

   From a command window (`xterm` in Linux or `cmd` in Windows), cd to your project directory and run the program like this:

   Debug/*projectName*

   replacing *projectName* by the project name that you chose, and, if you are running in Windows, using a backslash instead of a slash.

   The results should look familiar.

9. Some programs accept options from the command line instead of via direct prompts. This one can be run that way as well. Try running it as

   Debug/*projectName* 50

   You can accomplish the same thing when launching programs from Eclipse as well. From the Run menu, select "Run Configurations...". You should see that a C++ Application configuration named the same as the project is selected. (If

not, select “C/C++ Application” in the left column and then click the “New Launch COnfiguration” button at the top of that column.) On the right, click on the Arguments tab then enter 50 into the Program arguments box. Click Run. The arguments you have supplied will be used each time the program is run until you change them or create a separate run configuration.

10. You can now exit from Eclipse, or play around a bit to get more familiar with it.

3 Debugging

Next we turn our attention to debugging.

There are certain common functions that we expect an automatic debugger to provide:

- If a program crashes, the debugger will pause or freeze execution at the moment of the crash.
- Debuggers will allow you to set *breakpoints*, locations in the code where you want the debugger to pause the program whenever execution hists one of those locations.
- When you are paused, either by a crash or at a breakpoint, you can
  - examine the source code at the paused location, and the code of any functions that were called to get you there, and
  - Examine the values of variables at the moment of the pause.
- When you are paused at a breakpoint, you can

- Allow the execution to move forward in small steps.

  Most debuggers will have commands allowing you to step forward to the next statement in the same function where you are paused (often called "next"), or to try and step forward to the statement but, if any other functions are being called, to stop first at the start of those functions calls (often called "step"), or to run to the end of the current function in which you are paused (often called "finish").

  Some debuggers will have other options, such as stepping from one machine-code instruction to the next, but these are not used so often.

- Resume normal execution, running normally without stopping until the program crashes, ends normally, or until another breakpoint is hit (often called "continue").

The **gdb** debugger provides these functions for C++ code compiled with the gcc/g++ suite. The basic interfaces of **gdb** leaves a lot to be desired, however, so we usually rely on an IDE to provide a more convenient interfrace to **gdb**.

3.1 The Eclipse Debugger

Eclipse also offers good support for debugging.

1. Launch the Eclipse IDE. You will probably find that your former Eclipse project is already open and waiting for you. Eclipse tends to keep projects open until you close them. (If you leave many projects open, Eclipse will take a long time to start up because it begins by recompiling all open projects.)

2. Look at the file `findPrimes.cpp`. Near the bottom, you will find two statements that look like this:

```
// bool* theSieve = NULL; // error
bool* theSieve = new bool[maxNum]; // correct
```

Comment out the second statement and remove the comment markers from the left of the first statement. We are again deliberately injecting a bug into the program for the purpose of illustration. Compile the resulting program.

3. Via the Run menu, check the Run Configurations... You should find that there is an existing entry for a C++ application named "progdevx". Select it and check the Arguments. It probably recalls the integer you entered there earlier. If not, put a small integer, say, 50, in there. Click on Run and watch the program crash. (The crash can be quiet – it may simply stop running with no output at all.)

4. Next, run the program in the debugger. From the Run menu, select "Debug" or click the insect button. If you are asked about launchers, select "Using GDB...Create Process". The program starts running, but pauses at the first statement in main. Above the source code, you can see the call stack on th left and the local variables and parameters on the right. Click the "Step Over" button to move forward one statement. Note that maxNum changes value and is highlighted in the data display to show you what has changed. Clicking on any of the variables in the data display will cause more detailed info about it to appear at the bottom of that pane.

5. Click on the Resume button. The program runs until the crash. In the call stack area, you can see that it says that execution has been "suspended" because the program received a "signal". That "signal" is the segmentation fault that we have seen before, Click on the various entries in the call stack to see the editor window change to the call locations. Note that the data display changes to show the variables in each function that you have selected.

6. Click the Terminate button (the red square) to stop the execution of our program.

   Use the editor to repair the bug that we injected, and recompile. Run the program in the normal fashion, and observe that it completes correctly.

7. Find the body of the findPrimes function, and locate the line that writes to *cout*. Set a breakpoint at this line by double-clicking in the left margin alongside that line. Click the insect/bug button to start the debugger. You should stop at the beginning of main(). Click Resume, and execution should stop at your new breakpoint.

8. Looking at the code just in front of your breakpoint, you can see what the value of *message* is supposed to be. Check its value in the data display. You'll have to expand it at least once, because the version of Eclipse currently on our servers does not support the **gdb** pretty-printing option. (The latest version does, so this may change on our servers soon.)

   If you want to try the pretty-printing option (on our Linux servers), copy `~zeil/.gdbinit` to your project directory and restart Eclipse. This should improve the view you are given of *std::string* values as well as of the various `std::` containers such as *vector* or *list*.

9. Remove the breakpoint from this statement by double-clicking on the small blue breakpoint marker. "Click the "Step return" button to finish execution of this function, stopping as soon as we return to the caller. Take note of how the Watches and Call Stack windows are updated.

10. Exit from Eclipse.