“Compiled” Language Support in Eclipse

Chris Recoskie, committer -- CDT Project

Currently it can be said that there is no generic support for compiled languages in Eclipse. Certainly the platform APIs can be extended to provide the required functionality (*à la* CDT) but this would require a certain amount of duplication of work between the enablers of any given languages (i.e. the people that build a toolkit of some sort that plugs into Eclipse to provide support for their language of choice).

More often than not, what actually happens is that a compiled language enabler (e.g. the Fortran project) looks at the CDT and says to themselves “Well, CDT does 90% of what I need to do, so I’ll just reuse it.” Unfortunately the CDT was designed from the get-go, as the name implies, to be a set of C Development Tools, and for the most part, any sort of nod to multi-language support has only come in recent history, and as somewhat of an afterthought. As a result, a language enabler has an extremely difficult time trying to provide a good user experience for the consumers of their “product” if they wish to stick to the rule of “extend, do not modify.” Given the options of either providing a poor user experience or forking the CDT, the choice for these enablers becomes clear, and they proceed to fork away.

All of this seems a shame when there is so much in common between the various compiled languages out there in terms of tooling requirements. The experience for the user as they edit, build, and debug a compiled language project ought to be relatively similar regardless of the language in question. In fact, there is likely much overlap with non-compiled languages as well. The following is a summary of the “state of the union” for each of these three areas as I see them.

Editing

End user developers edit textual source files which contain their code. Aside from the standard text editor type operations which we will take as a given, users like to be able to format their source code for better readability, insert boilerplate comments, comment/uncomment blocks of source code, auto insert braces, etc. I.e., there are a certain set of what I would call text-editor-like operations which coders like to use on their source code, which although useful in a programming context, require a minimal amount of insight into the actual structure of the programming language which they are using. It is fairly easy for instance, to provide a generic editor which allows the user to configure preferences for the comment characters that the language in question will use, and to use that preference to allow the user to comment and uncomment blocks of code with the click of a button.

Users also wish to have more complicated features which require a somewhat more intimate knowledge of the language they are using. Generally speaking there are well known standards for most of these languages, so the grammars describing these

languages are freely available. As such, one expects that it should be possible for a common editing subsystem to somehow be able to take the grammatical description of a given language and be able to automatically parse the source files as the user edits them, providing useful functionality such as syntax highlighting, error detection, content assist, structural analysis (e.g. populating the outline view), refactoring, etc. In theory at least, language enablers could simply provide the grammatical descriptions of their languages, rather than reinvent the wheel and rolling their own abstract syntax tree implementation, parser engine, refactoring engine, etc. Such a system would be the holy grail so to speak of generic editor support.

Building

The source files the user produced are then processed via some sort of command line tools to produce debuggable executables for the targeted software/hardware system platform. Most of these tools behave in a relatively similar fashion. A compiler takes the source files and compiles them into object files, which are later consumed by a linker and potentially combined with other linkable libraries into a final executable. Although the details of course differ, most individual build tools (e.g. the compiler) behave in a relatively similar fashion across vendors. The tools have options (e.g. optimization levels, include paths, etc.) which the users wish to manipulate to tweak their build process.

It is important to note that as far as building is concerned, there is a lot of leeway in what we can consider a compiled language. For instance, consider a project consisting of assembly language sources, which are assembled into object files, and linked into a final executable. Technically, no compilation actually occurs, since the tool used is an assembler, not a compiler, but from the builder's point of view, it looks a lot like a compiler.

There exists within the CDT already a pair of relatively language agnostic builders that could be potentially reused to provide the basis for a generic build system: Standard Make, and Managed Build.

Standard Make provides bare bones make driven functionality – i.e. the user can specify a makefile and a make utility, and from that point on when they hit build CDT will just invoke make with their makefile, and disclaims any knowledge of the magical process by which the project will actually be built.

Managed Build takes a somewhat more hands on approach. The idea is that the user need know nothing about make, or of makefiles. Instead, Managed Build generates the makefile for them, and the user manipulates their project resources using the standard Eclipse GUI. The Managed Build System (MBS) provides a generic extension point framework to allow ISVs to customize which build utility to use (e.g. GNU make, nmake, the OS shell, etc.), how to generate a "makefile" for the builder (which might or might not be an actual makefile depending on whether an actual make utility is used), what the

tools to invoke are and in what order, the options those tools take and how to represent them in the GUI, etc. MBS is generally agnostic about what the tools are and what they actually do, and only requires the ISV to specify enough information in their toolchain definition so that the tools are invoked properly and in the right order, allowing the generated data to flow properly from tool to tool.

As previously stated, for the most part the builders in CDT are language agnostic. Really when you get right down to it, there is not much difference between building a Fortran project and a C project. Most of the C/C++-isms that exist are by virtue of the builders being bundled with CDT in the first place. For example, CDT defines two project natures, the `cnature` and the `ccnature` (which it uses to identify C and C++ projects respectively), and in order to create either one of a Standard Make or a Managed Build project, the user must inherently pick one of these natures. Also, the builders are obliged to provide a certain amount of data to other parts of CDT (e.g. user specified include paths to the DOM parser) so that the user has a nicely integrated experience. None of these pieces of the puzzle are core to the functionality of the builders themselves however, so in theory there would be nothing really stopping us from moving the builders outside of CDT in order to do away with these and other C/C++-isms.

Debugging

Once the user has built their project, they will presumably wish to debug it at some point. The debugging waters get a bit murky, but there is the potential for a shared implementation across most compiled languages. Most compiled languages are procedural in nature so they share common traits such as the notion of an execution stack, global vs. procedure/function scope variables, aggregate types, access to machine resources such as memory or registers, etc.

What is more, in terms of implementation details, much of the plumbing is entirely identical. To debug a program, you must load it and launch it. You must load the executable's debug symbol table (usually in some standard format such as DWARF) to figure out where functions, variables, and other symbols are stored in memory, and what lines of source code the various symbols correspond to. When execution of the debugged program is halted, the debugger uses the symbol information to figure out what file and which line of that file to display the program counter icon at. Hitting the step over button involves creative use of temporary breakpoints which are transparent to the user. These types of operations, and others, remain relatively constant across compiled languages. For many core debugging operations, what is important is not so much the details of the language in question, but rather understanding the relationship between the machine code and the source code it originated from. So, one wonders what it would involve to create a language neutral, compiled language debugger implementation under the platform debug APIs, thus removing the need for compiled language enablers to reinvent this wheel themselves.

There is already the potential for a shared debugging implementation based on the debugger in CDT. The GDB command line debugger already supports a multitude of different languages, and the stock debugger implementation in CDT already uses GDB for a debugging engine. It is possible that this implementation could be moved outside of CDT to provide a language-agnostic GDB debugger implementation which could be utilized by various language sub-projects.