**Developing custom plug-ins for the Vim editor**

Arpan Sen November 09, 2010

**Introduction**

You'll be using Vim version 7.2 and Perl version 5.8.

**Syntax highlighting**

**Listing 1. Keywords from your custom programming language**

foreach if then else elsif while repeat until disable integer unsigned signed byte

always initial

|  |  |
| --- | --- |
|  |  |
|  |  |

Vim associates some word to be a keyword using the syntax keyword <group name> <keyword list>

So, for your custom language, use the pseudo-code [Listing](#page2) .2

**Listing 2. Defining keywords in Vim**

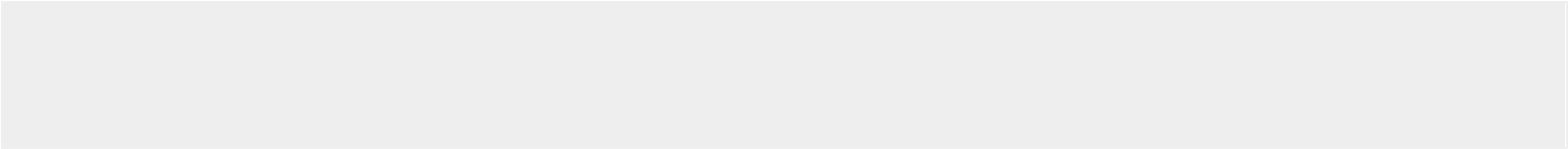


syntax keyword group1 foreach if then else elsif while

repeat until disable integer unsigned signed byte always initial

Next, save this fi*lang*eas*.vim* under $HOME. Now, edit a small snippet of code in your custom language (see [Listing](#page2) )3.

**Listing 3. Code in the custom programming language**



integer k=0;

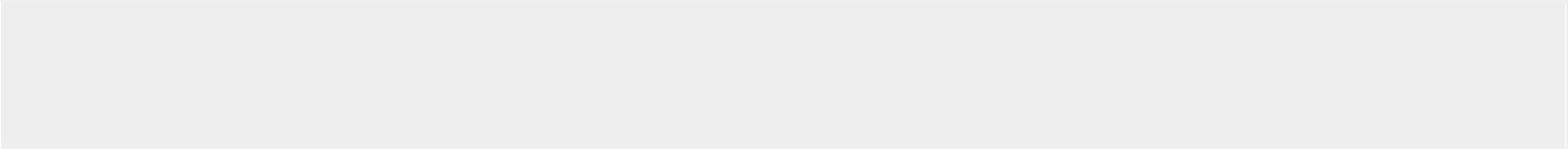
repeat (k < 3) begin

print “hello world” + k + “\n”; k = k + 1;

end

In the Vim editor, load lang.vim as :source $HOME/lang.vim and you are done. But wait, t problem, nothing happens. Even though you specified the syntax you did not provide input how it should be highlighted [Listing.](#page2) 4shows an improved version of the lang.vim file.

**Listing 4. Improved version of lang.vim that supports syntax highlighti**



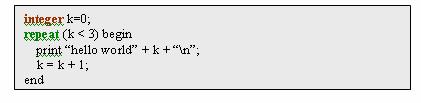
syntax keyword type1 integer unsigned signed byte syntax keyword statement1 foreach if then else elsif

while repeat until disable always initial highlight link type1 Type

highlight link statement1 Statement

Reload lang.vim, and the code from [Listing](#page2) 3is now syntax highlighted  [Figure(see](#page2) )1. In Figure 1 below, the keywords*integer* and *repeat* are highlighted.

**Figure 1. Highlighted syntax in the custom code**



What exactly did you do [Listingin](#page2) ?4 There are two things to understand:

* The user would typically expect the statements in theif-then-else,repeat, and so

on) highlighted differently from the dataintegertypes,byte) for better readability. So, you

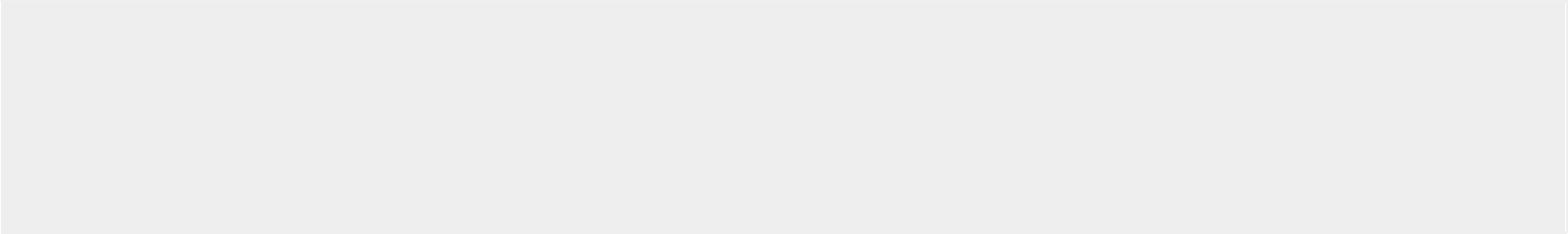
split the syntax into groups with appropriate contents:type1 containsgroup theinteger, unsigned,signed, andbyte keywords.

* Vim has predefined syntax groups Type,Statement,Comment, andIdentifier that come with their specific color schemeshighlight command associatestype1 with Vim’sType group so that the same color scheme reflects for keywordsbyte. like

**More syntax support**

You would probably like your language to be case-neutralintegerthandtINTEGER both get highlighted. You’d also like to add support//C++-forstyle comments [Listing.](#page3) 5shows the modified lang.vim file.

**Listing 5. Improved version of lang.vim that supports syntax highlighti**



syntax case ignore

syntax keyword type1 integer unsigned signed byte syntax keyword statement1 foreach if then else elsif

while repeat until disable always initial syntax match comment1 /\/\/.\*/

highlight link type1 Type highlight link statement1 Statement highlight link comment1 Comment

The syntax case ignore statement takes care of the case neutrality. You cannot handle comm using keywords, so you need a regular expression that you can then associate with the Vi

Comment group. You define the regular expressionsyntax match <identifier> /<pattern>/. In between the start and end forward /slash),you( defined the pattern\/\/.\*, which signified anything that started //withcontinued to the end of the line. So, the code [Figureshown](#page3) 2infrom your custom language has the proper highlighting.

**Figure 2. Support for comments and case independence**



**Custom coding standard support**

You can easily extend the custom plug-in to handle organization-specific coding standard are some typical guidelines:

* No tabs in the code.
* No variable name should be longer than 14 characters.
* Do not have more than 80 characters on a single line.
* Functions may not be longer than 100 lines.

**No tabs in the code**

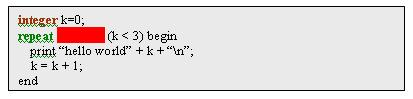
Let's begin with the simplest guidelines, no tabs in the code. You simply definetaban iden then associate that identifier with Vim's predefinedErrortag:



syntax match identifier1 “\t” highlight link identifier1 Error

So, what happens if you do have tabs in your [Figurecode?](#page4) 3shows a revised version of the code in [Figure](#page3) 2with tabs.

**Figure 3. Vim highlighting the existence of tabs in red**



In Figure 3, the tab just*repeat*afteris highlighted in red; a clear flag to the user that somet wrong.

**No variable name longer than 14 characters**

Support for variable name lengths requires more understanding of the use of regular expr in syntax match:



syntax match longword1 “\w\{14,}” highlight link longword1 Error

Here,\w defines the character [0-9A-Za-z\_]—that is, any digit, alphabetic character (uppercase or lowercase), or underscore)is allowed(. The next sequence is

\{14,}, which means that at least 14 consecutive occurrences need to be matched. So, this\_is\_a\_REAL\_long\_word1 will result in a highlight, as the identifier length is greater whilethis\_is\_ok\_2 is going to do just [Figurefine.](#page4) 4shows how an error situation looks.

**Figure 4. Variable names longer than 14 characters are flagged in red**



In Figure 4, the variable*thisis\_a\_REAL\_long\_word1* is highlighted in red (again as per Vim’s default color profile) warning the user that something is wrong.

**No more than 80 characters on a single line**

Having more than 80 characters on a single line adds to the clutter and makes reading di You would again be usingsyntax match to define an identifier for this regular expression an toError. Figuring out the regular expression should not be difficult:^)signifiesThecaratthe( start of a line; the dollar$ )signsignifies( the end of a line; and anything in between should be lo than 80 characters for an error match. Note that the.) signifiesperiod( a match for any character other than end-of-line:



syntax match longline1 “^.\{80,}$” highlight link longline1 Error

[Figure](#page5) 5shows a code line that exceeds 80 characters in length and is therefore highligh

Developing custom plug-ins for the Vim editor Page 4 of8

ibm.com/developerWorks/ developerWorks®

**Figure 5. Rules to prevent line length from exceeding 80 characters**

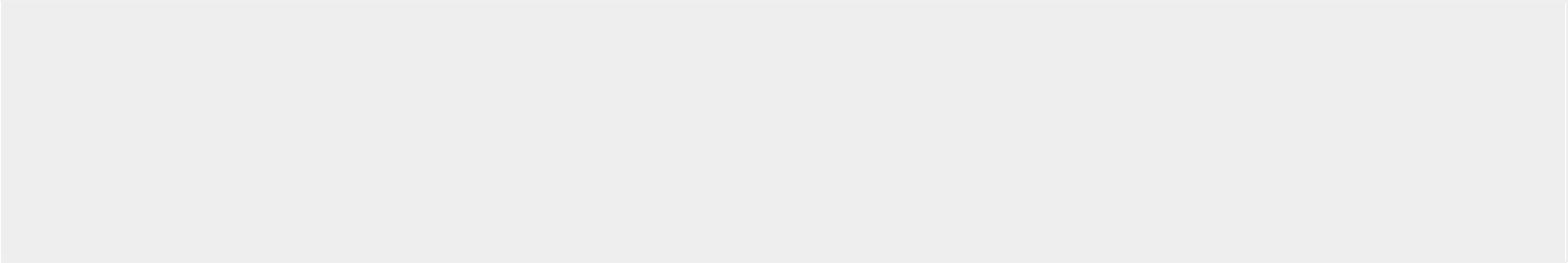


Since we exceeded 80 characters thanks to the complex formula, in Figure 5, Vim highligh the whole line in red. The highlight will go the moment we enter a backspace and the lin reduces to 79 from 80.

**No functions longer than 100 lines**

The last in the list of custom coding conventions is that function length must always be 100 lines [Listing.](#page5) 6shows a function defined in your custom programming language.

**Listing 6. A function defined in the custom programming language**



function f (int k, int l) returns float begin

f = k \* l;

for (int i=0; i<10; i++) begin

f += sqrt(k) \* sqrt(l);

end

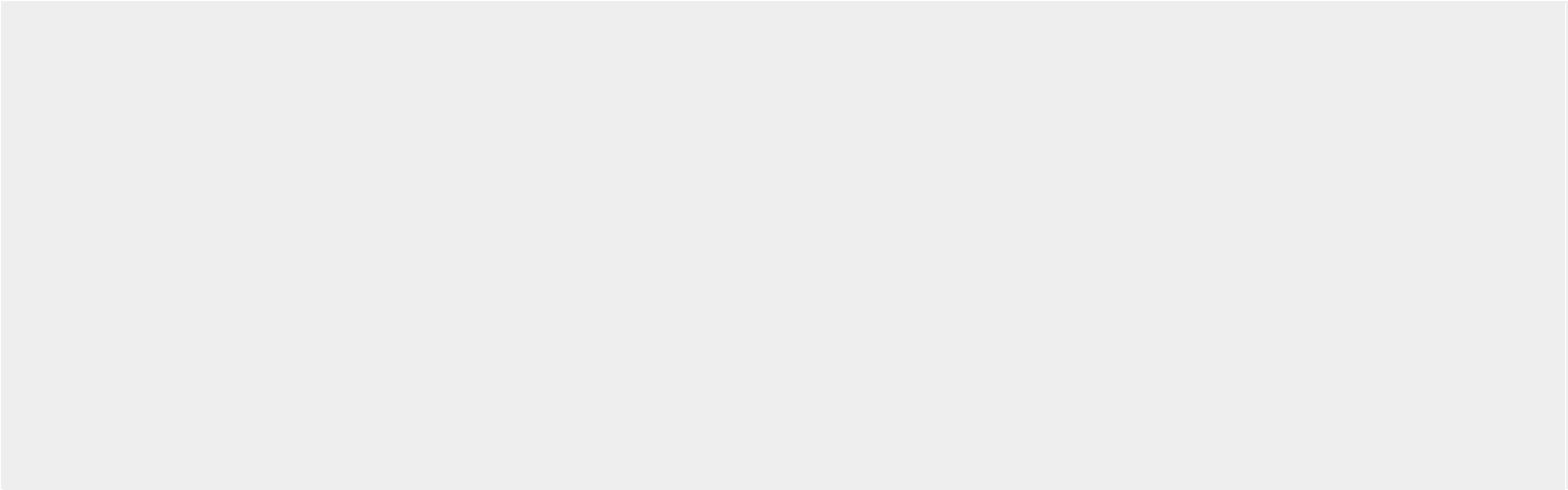
return f + 2; endfunction

It's probably easier (and definitely faster) to have the entire function passed to Perl the number of lines than to devise a complex regular expression or call Vim predefined i functions. The next section has the details.

**Using external scripting languages to create a Vim plug-in**

a Vim plug-in that displays an error message if the length of any function is greater than 100

**Listing 7. Creating a custom plug-in for Vim using Perl**



perl << EOF sub checksize

{

my $count = 0;

my $startfunc = 0;

my $filelen = scalar @\_; while ($count < $filelen)

{

if ($\_[$count] =~ /^function/)

{

$startfunc = $count;

}

elsif ($\_[$count] =~ /endfunction/)

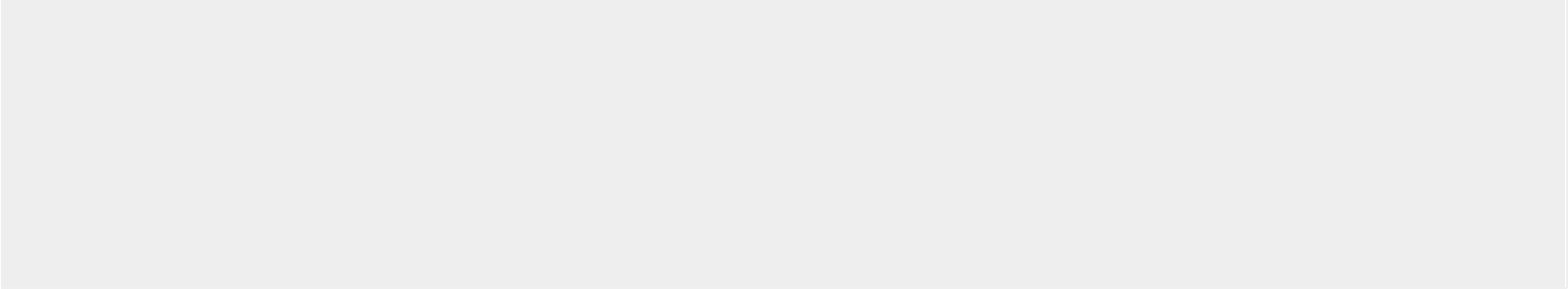
{

if ($count - $startfunc > 100)

{

Vim::Msg($\_[$startfunc], "Error");

}



}

++$count;

}

}

EOF

function! L1( )

perl checksize($curbuf->Get(1..$curbuf->Count())) endfunction

All of this is coded in the same lang.vim file you used earlier. Here are the nuances of

1.You embed Perl code inside Vim script using markers. These markers could have any na and may not be in all uppercase characters [Listing.In](#page5) 's7perl << EOF … EOF,EOF was the

marker used. Make sure that the secondEOF begins at the first column of a line. The mark does not have to be namedEOF—any name would do—but the first-column rule is sacrosanct

2.The entire contents of the file are passed to the Perl code. The checksizesubroutine traverses over the whole file (as part@ implicit array in Perl) and keeps checking fo function lengths. When it encounters the function, it sets a counter to 0; when it h on theendfunction keyword, it checks whether the counter is greater than 100. If the length exceeds 100, an error message is displayed.

3.You cannot use the usual Perl print routines to display the error message, because y the message to be displayed inside Vim. Vim provides a useful interface to Perl, the which are availableResourcesin.Vim::Msg displays the message inside the editor window.

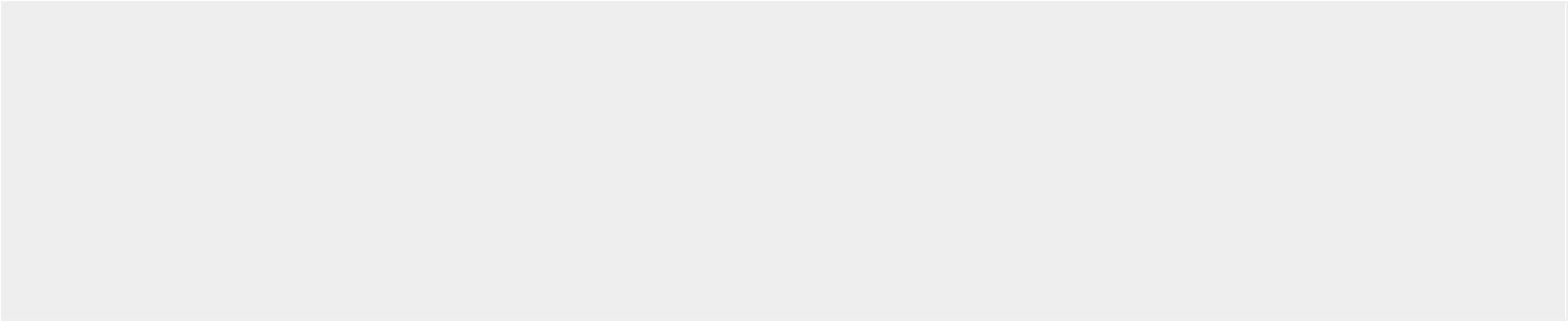
[Listing](#page5) ,7 you display the first line of the offending function. The secondVim::Msgargument t is the type of information that is displayed:errorimplies that this information is highlighte red.

4.You define a function in Vim that passes to the Perl code the $curbuf->Count( ) tells you how many lines there are in the current$curbuf->Get(<line1>..<line2>)

returns the text between the lines specifiedline1 andbyline2 . In the script, you are passing the contents from line 1 to last line of the current buffer. Now, in:call L1(): type You should immediately see the offending function(s) listed.

**Compiling sources from within Vim**

**Listing 8. Really messed up C++ code**



#include <iostream> using namespace stdl

class mytags { public:

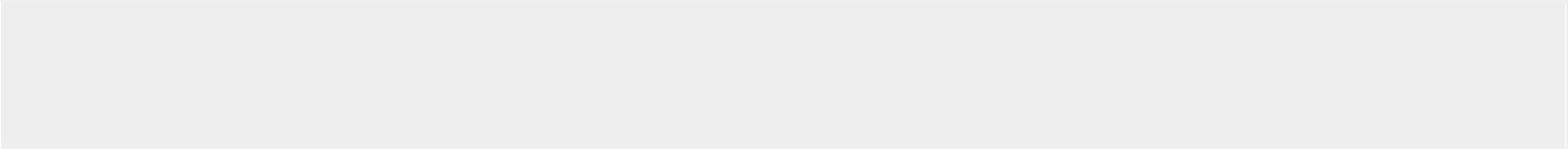
int getid(int id=0); void setid(int)

protectd: list<int> tags;

const list<int>::iterator tag\_i;

}

**Listing 9. Mapping the F3 key to compile and display errors inside edit**



function! build() make

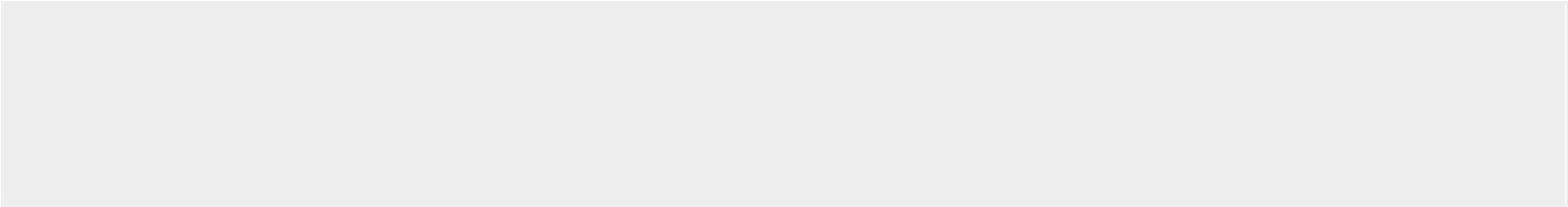
cl “list the errors endfunction

map <F3> :call build()<CR>

Press the F3 key in ESC mode to compile the sourcesbuild() function callsmake from inside Vim, and then invokescl, which displays the errors. To go to the first:cfirst intypeESC mode; to go to every subsequent error,:cnuse;to go to the last error,:clasttype. Note that by

default, Makefile is assumed to be in the same folder as the sources. That said, none of necessary, because you can modify build() function from [Listing](#page7) 9to go to the folder in which theMakefile resides. Also, it is easy enough to pass argumentsmake. [Listingto](#page7) 10clarifies the point.

**Listing 10. An improved Vim script for building sources using make**



function! build()

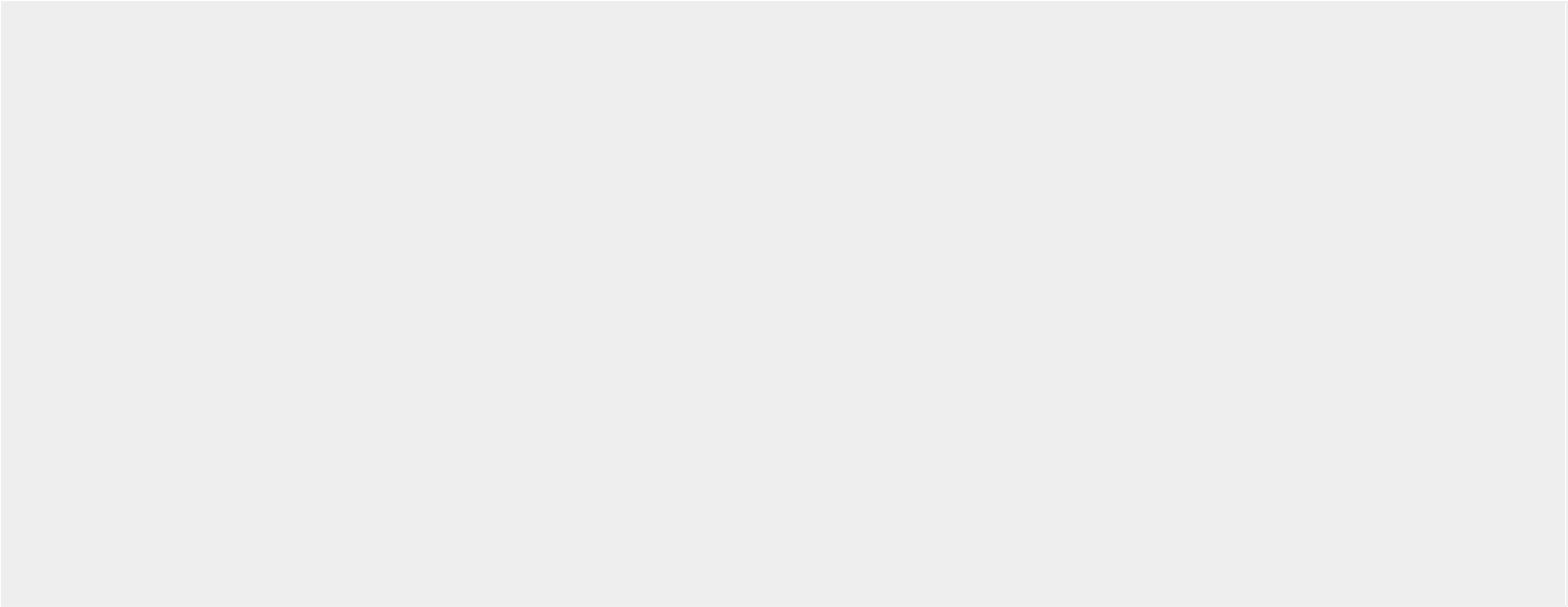
cd /home/arpan/ibm/scripts “go to the folder where Makefile is make CC=g++

cd /home/sources “back to sources cl “list the errors

endfunction

map <F3> :call build()<CR>

**Listing 11. Build errors inside the Vim environment**



#include <iostream> using namespace stdl

class mytags { public:

int getid(int id=0); void setid(int)

protectd: list<int> tags;

const list<int>::iterator tag\_i;

}

t.cpp:4: error: expected namespace-name before "class" t.cpp:4: error: `<type error>' is not a namespace t.cpp:4: error: expected `;' before "class"

t.cpp:8: error: expected `;' before "protectd"

t.cpp:10: error: ISO C++ forbids declaration of `list' with no type t.cpp:10: error: expected `;' before '<' token

t.cpp:6: error: expected unqualified-id at end of input t.cpp:6: error: expected `,' or `;' at end of input Press ENTER or type command to continue

**Installing a syntax file**

Create a folder named*syntax* under $HOME/.vim, and copy the custom plug-in to it. If your c language is called*ml2,*, say, then name this*ml2*file*.vim.* Then, edit $HOME/.vimrc and add the

linesyntax on. That's it: Now, whenever you open a file with the*ml2*extensioninvim, the syntax is automatically highlighted. This behavior does not include explicit function calls; it have key mappings for quick custom code checks, such as subroutine length.