

```
// =====  
// 版本信息：0.3  
// 作者姓名：杨新涛  
// 电子邮件：yangyang.gnu@gmail.com  
// 更新时间：2012-12-9 22:52:46  
// 版权信息：本文版权归杨新涛所有。非商业转载，请保留本文档信息；商业转载，须经本人同意  
// =====
```

拼装的艺术：vim 之 IDE 进化实录

Ken Thompson 告诉我们——“一个程序只实现一个功能，且做到极致，多个程序协作实现复杂任务”——这是 unix。是滴，这种哲学在 linux 上随处可见，比如，vim 与她的插件们（白雪公主与七个小矮人 -_- \$）。下面开始我们的 vim 之 IDE 进化之旅吧。

这个时代，上规模的软件项目已不可能用简单的文本编辑器完成，IDE 是必然选择。linux 下 IDE 大致分为两类：“品牌机”和“组装机”。“品牌机”中有些（开源）产品还不错，比如：codeblocks、anjuta、SciTE、netbeans、eclipse 等等，对于初涉 linux 开发的朋友而言是个不错的选择（我指的是 codeblocks），但对于老鸟来说总有这样那样的欠缺。听闻 linus torvalds 这类大牛用的是类 emacs（准确的说是 microemacs）和一堆插件拼装而成的 IDE，为向大牛致敬，加之那颗“喜欢折腾”的心，“组装机”是我的选择。首要任务，选择编辑器。

linux 上存在两种编辑器：神之编辑器——emacs，编辑器之神——vim。关于 emacs 与 vim 孰轻谁重之争已是世纪话题，我无意参与其中，在我眼里，二者都是创世纪的优秀编辑器，至少在这个领域作到了极致，它们让世人重新认识了编辑操作的本质——用命令而非键盘——去完成编辑任务。好了，如果你不是 emacs 控，不要浪费时间再去比较，选择学习曲线相较平滑的那个直接啃 man 吧——vim 不会让你失望的。

对于 vim 的喜爱，我无法用言语表述，献上一首湿哥哥（-_-#）以表景仰之情：

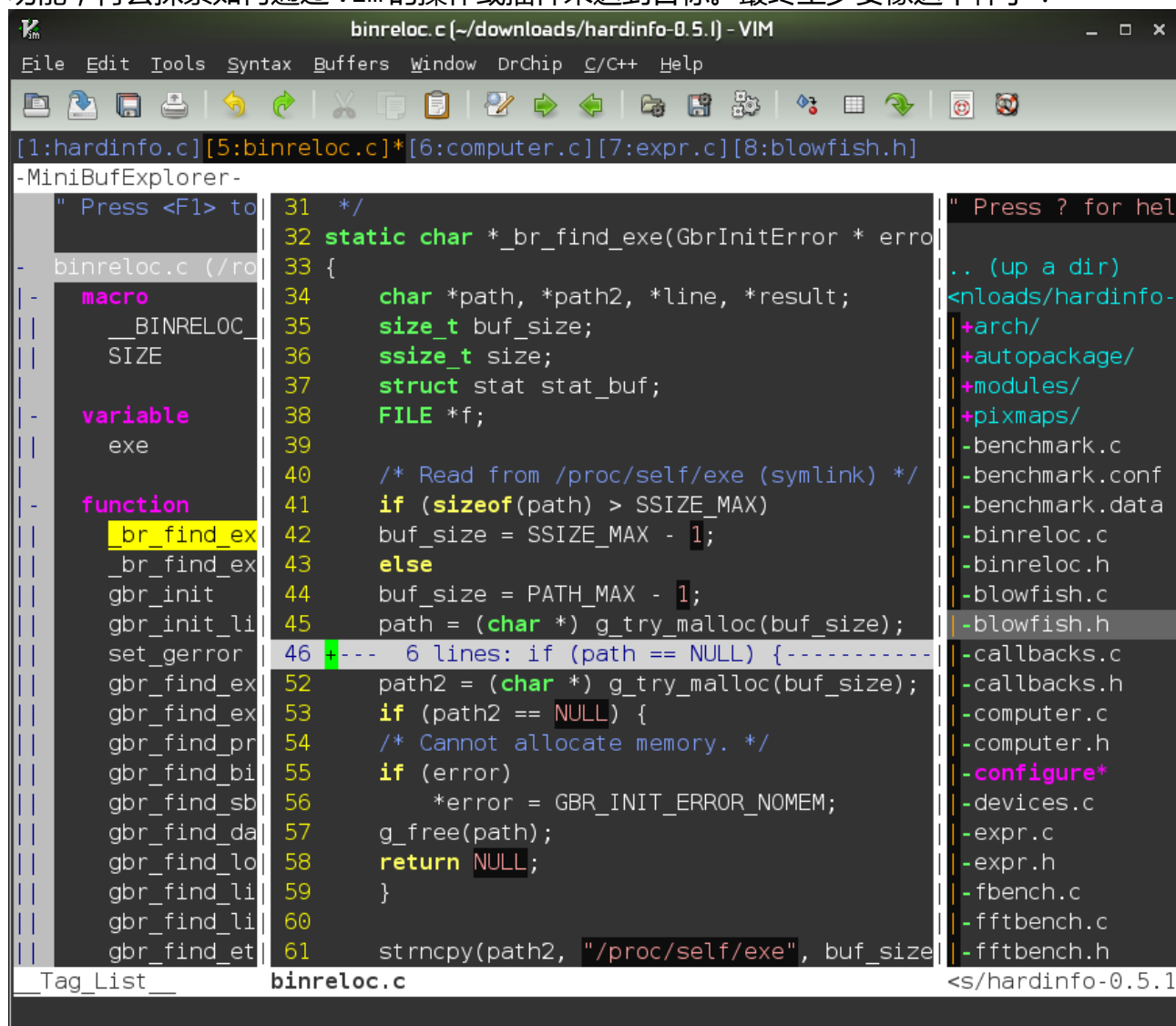
*vi 之大道如我心之禅，
vi 之漫路即为禅修，
vi 之命令禅印于心，
未得此道者视之怪诞，
与之相伴者洞其真谛，
长修此道者巨变人生。*

——作：reddy@lion.austin.com，

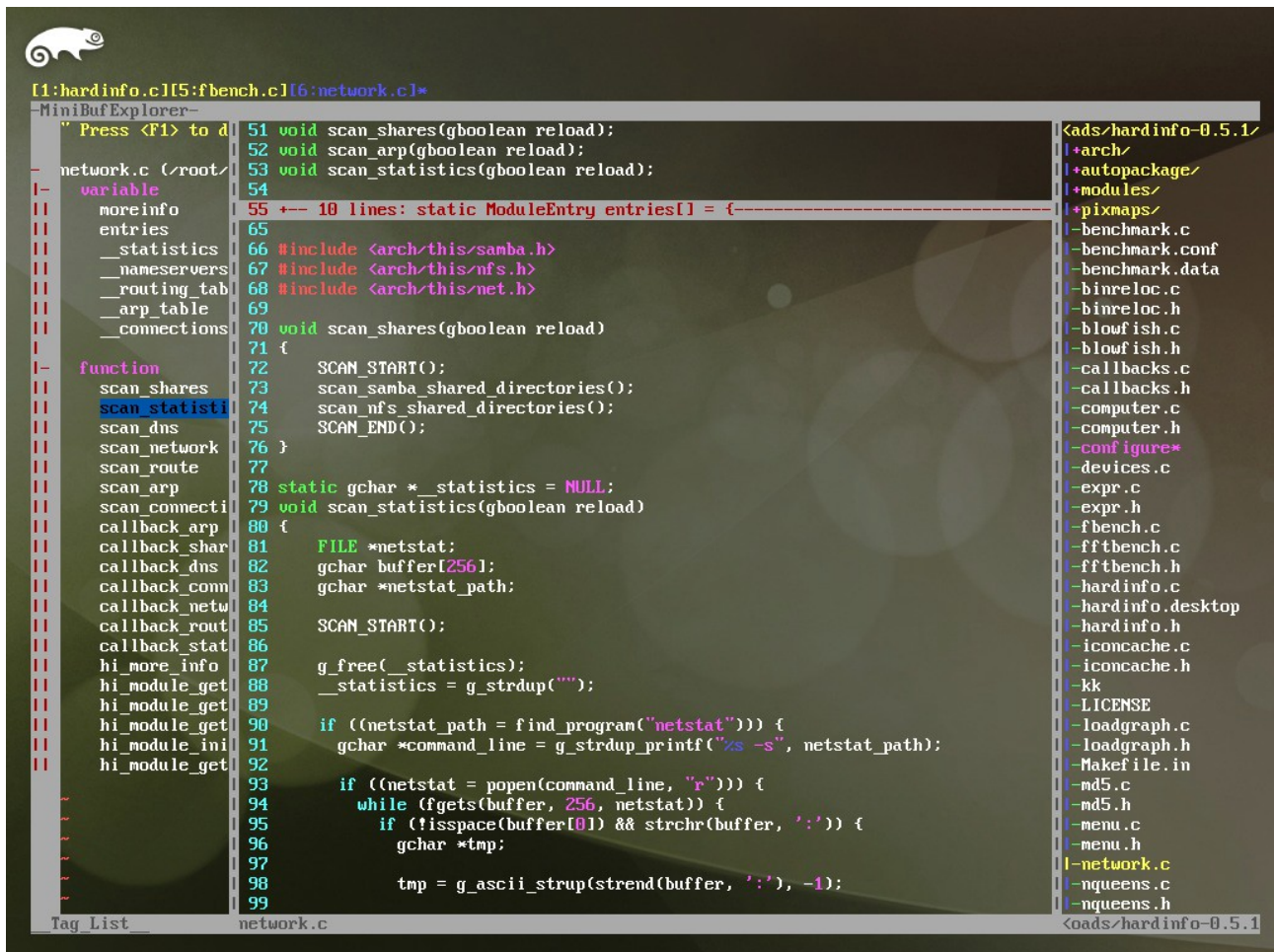
——译：yangyang.gnu@gmail.com

OK，言归正传，说说 vim 用于代码编写提供了哪些直接和间接功能支撑。vim 联机手册中，50% 的例子都是在讲 vim 如何高效编写代码，由此可见，vim 是一款面向程序员的编辑器，即使某些功能 vim 无法直接完成，借助其丰富的插件资源，必定可以达成目标（注，推荐两份 vim 入门资料：《vim 用户手册中文版 7.2》、《A Byte of Vim v0.51 (for Vim version 7)》）。

我是个“目标驱动”的信奉者，本文内容，我会先给出优秀 C/C++ IDE 应具备哪些功能，再去探索如何通过 vim 的操作或插件来达到目标。最终至少要像这个样子：



(图形环境下 IDE 总揽)



(纯字符模式下 IDE 总揽)

在介绍功能 IDE 应具备的功能之前，先说说 vim 的两个“必知会”：

1、vim 的可扩展性——插件。

vim 有一套自己的脚本语言，通过这种脚本语言可以实现与 vim 交互，达到功能扩展的目的。一组 vim 脚本就是一个 vim 插件，vim 的很多功能都是通过其插件实现，在其官网上有丰富的插件资源，任何你想得到的功能，如果 vim 无法直接支持，那一般都有对应的插件为你服务，有需求时可以去逛逛。

vim 插件目前分为两类：*.vim 和 *.vba。前者是传统格式的插件，实际上就是一个文本文件，通常 *someplugin.vim*（插件脚本）与 *someplugin.txt*（插件帮助文件）并存在一个打包文件中，解包后将 *someplugin.vim* 拷贝到 *~/.vim/plugin/* 目录，*someplugin.txt* 拷贝到 *~/.vim/doc/* 目录即可完成安装，重启 vim 后刚安装的插件就已经生效，但帮助文件需执行 *helptags ~/.vim/doc/* 才能生效。传统格式插件需要解包和两次拷贝才能完成安装，相对较繁琐，所以后来又出现了 *.vba 格式插件，安装更便捷，只需在 shell 中依次执行如下命令即可：

```
vim someplugin.vba
```

```
:so %  
:q
```

另外，后面涉及到的各类插件，只介绍了我常用的操作，有时间，建议看看它们的帮组文档（`:h someplugin`）。

2、vim 的灵活性——自定义。

不论是 vim 窗口外观、显示字体，还是操作方式、快捷键、插件属性均可通过编辑配置文件将 vim 调教成最适合你的编辑器。vim 的全局配置文件位于 `/etc/vimrc`，它控制着所有用户下的 vim，局部配置文件位于 `~/.vimrc`，如果调整了该文件，相关变动仅对当前用户有效。

再说说插件帮助文档和 vim 自身帮助文档中经常出现的“`<leader>`”。要正常调用各项插件功能（甚至 vim 自身的很多快捷操作），先得输入一个“前缀键”，通过“前缀键”告诉 vim 说现在用户输入的是快捷键而非普通字符。正因如此，“前缀键”是 vim 使用率较高的一个键（最高的当属 `Esc`），选一个最方便输入的键作为“前缀键”，将有助于提高编辑效率。我用的是分号，请将“前缀键”配置信息加入 `.vimrc` 中：

```
" 定义快捷键的前缀，即<Leader>  
let mapleader=";"
```

我的配置文件参见附一，每个配置项都有对应注释，可根据你自己情况按需择取。

【添加与删除注释】

注释时到每行代码前输入 `//`，取消注释时再删除 `//`，这种方式不是现代人的行为。IDE 应该支持对选中文本块批量（每行）添加注释符号，反之，可批量取消。本来 vim 通过宏方式可以支持该功能，但每次注释时要自己录制宏，关闭 vim 后宏无法保存，所以有人专门编写了一款插件，其中部分功能就是快速注释与反注释。

- 插件名：NERD Commenter

- 常用操作：

`<leader>cc`，用 CPP 语法风格注释掉选中文本块或当前行（NERD Commenter 根据编辑文档的扩展名自适应采用何种注释风格。如，文档名 `x.cpp` 则采用 `"//"` 注释风格，而 `x.c` 则是 `"/**/"` 注释风格）

`<leader>cu`，取消选中文本块或当前行的 CPP 语法风格注释

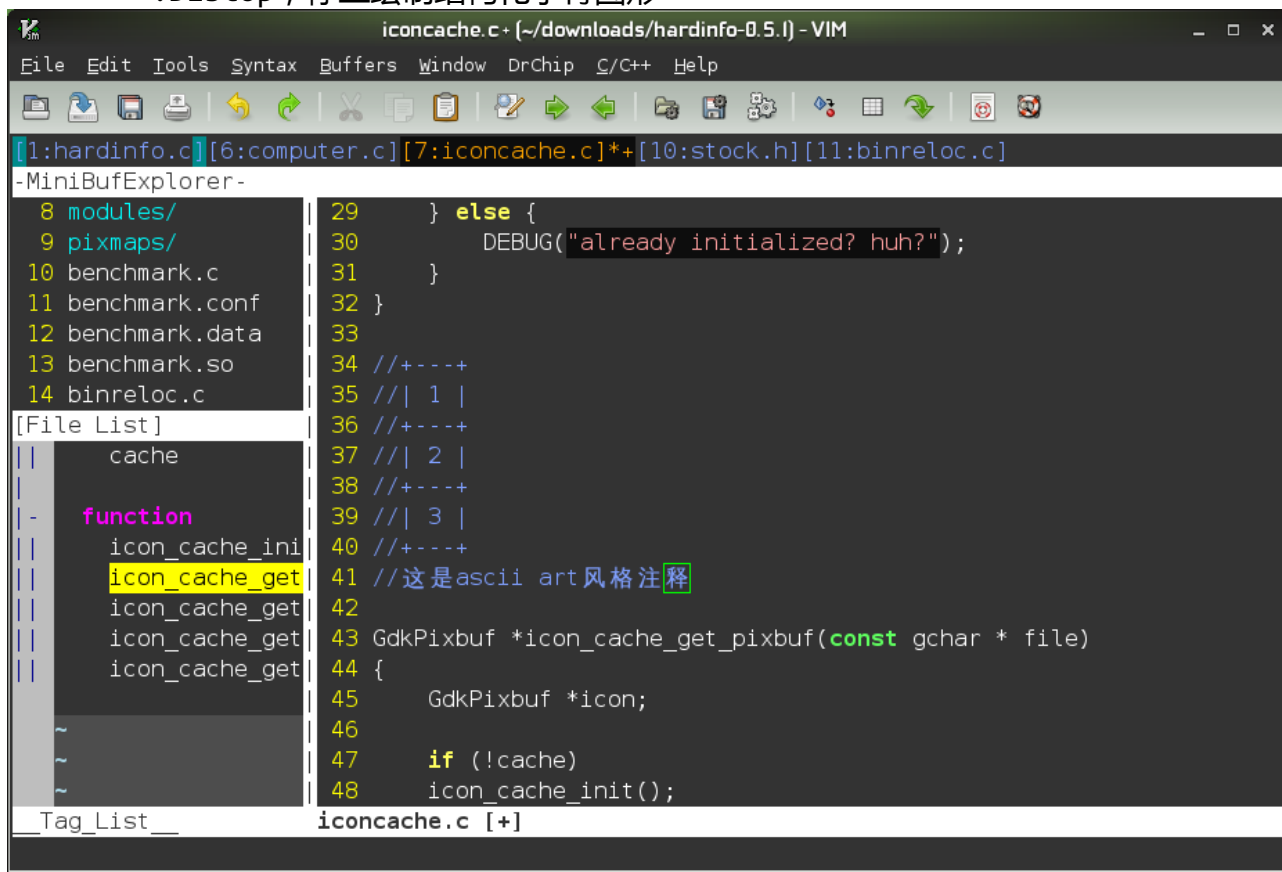
- 注意：由于 C 风格注释有“嵌套注释”风险，强烈建议即便 C 程序也用 CPP 风格注释。

此外，有时需要 `ascii art` 风格注释，推荐如下插件：

- 插件名：DrawIt.vim

常用操作：

- :Dlstart，开始绘制结构化字符图形，这时可用方向键绘制线条，空格键绘制或擦除字符
- :Distop，停止绘制结构化字符图形



```
iconcache.c+ [~/downloads/hardinfo-0.5.l] - VIM
File Edit Tools Syntax Buffers Window DrChip C/C++ Help

[1:hardinfo.c][6:computer.c][7:iconcache.c]*+[10:stock.h][11:binreloc.c]
-MiniBufExplorer-
8 modules/
9 pixmaps/
10 benchmark.c
11 benchmark.conf
12 benchmark.data
13 benchmark.so
14 binreloc.c
[File List]
| | cache
| |
| - function
| | icon_cache_ini
| | icon_cache_get
| | icon_cache_get
| | icon_cache_get
| | icon_cache_get
| |
| ~
| ~
| ~
Tag_List_ iconcache.c [+]
```

(ascii art 风格注释)

【全能补全】

“智能补全”是提升编码效率的杀手锏。试想下，有个函数叫 `get_count_and_size_from_remotefile()`，当你输入 “`get_`” 后 IDE 自动帮你输入完整的函数名，又如，有个文件 `~/yangyang.gnu/this/is/a/deep/dir/myfile.txt`，就像在 shell 中一样，类似 tab 键的东东自动补全文件路径那是何等的惬意啊！以上两个例子仅是我需要的补全功能的一部分，完整的补全功能应具备：1) 预处理语句、语法语句、语法关键字、函数框架补全；2) (标准库和自定义的) 函数名、变量名、结构名、结构成员、头文件名、文件路径。

第一类补全，实际上是通过快捷键插入代码片断，可借助 `snipMate.vim` 实现 (通过 <缩写词>tab 激活)。要写 do-while 语句只需简单的输入 “`do[tab]`”，要包括头文件输入 “`inc[tab]`” 即可出现 `#include <XX>`。

- 插件名：snipMate.vim

- 操作：缩写词[tab]

- 注意：所有模板位于\$HOME/.vim/snippets/目录，可按个人偏好设置。

c.snippets 和 cpp.snippets 分别为 C 和 C++ 相关的模板文件，我把个性化设置都配置在 cpp.snippets 中，为避免重复，已将 c.snippets 文件内容清空。平时，最让我头痛的字符莫过于 {}、""、[] 等这类结对符，输入它们之所以麻烦（低效），主要因为：A）要同时按住 shift 键，B）这些字符所在的键不是十个指头“触手可及”的区域（也就是你要眼睛看了才知道这些键的具体位置）。所谓高效输入结对符，应该是输入少量几个字母（没错，我写的是字母，不是字符）后 vim 自动补全结对符，而非是“人工输入一半 vim 输入另一半”。通过对 ~/.vim/snippets/cpp.snippets 适当扩充，可完美地解决这类问题。完整的 cpp.snippets 如下，可按需择取：

```
#=====
#预处理
#=====
# #include <...>
snippet inc
    #include "${1:Filename("$1.h")}"${2}
# #include <...>
snippet INC
    #include <${1:TODO}>${2}
#=====
#结构语句
#=====
# return
snippet re
    return (${1:/* condition */});
# If Condition
snippet if
    if (${1:/* condition */}) {
        ${2:TODO}
    }
snippet ei
    else if (${1:/* condition */}) {
        ${2:TODO}
    }
}
```

```

snippet el
    else {
        ${1:TODO}
    }
# Do While Loop
snippet do
    do {
        ${2:TODO}
    } while (${1:/* condition */});
# While Loop
snippet wh
    while (${1:/* condition */}) {
        ${2:TODO}
    }
# switch
snippet sw
    switch (${1:/* condition */}) {
        case ${2:c}:
            {
                ${3:TODO}
            }
            break;

        default:
            TODO
    }
# 通过迭代器遍历容器（可读写）
snippet for
    for (auto ${2:iter} = ${1:c}.begin(); ${3:$2} != $1.end(); ${4:++iter}) {
        ${5:TODO}
    }
# 通过迭代器遍历容器（只读）
snippet cfor
    for (auto ${2:citer} = ${1:c}.cbegin(); ${3:$2} != $1.cend(); ${4:++citer}) {
        ${5:TODO}
    }
# 通过下标遍历容器

```

```

snippet For
    for (auto ${2:i} = 0; $2 != ${1}.size(); ${3:++}$2) {
        ${4:TODO}
    }
# C++11 风格 for 循环遍历 ( 可读写 )
snippet F
    for (auto e : ${1:c}) {
        TODO
    }
# C++11 风格 for 循环遍历 ( 只读 )
snippet CF
    for (const auto e : ${1:c}) {
        TODO
    }
# For Loop
snippet FOR
    for (int ${2:i} = 0; $2 < ${1:count}; ${3:++}$2) {
        ${4:TODO}
    }
# try-catch
snippet try
    try {
        TODO
    } catch (${1:/* condition */}) {
        TODO
    }

#=====
#容器
#=====
# std::vector
snippet vec
    vector<${1:char}> v${2};
# std::list
snippet lst
    list<${1:char}> l${2};
# std::set
snippet set
    set<${1:key}> s${2};

```



```

# std::map
snippet map
    map<${1:key}, ${2:value}>      m${3};
=====
#语言扩展
=====
# Class
snippet cl
    class ${1:Filename('${1_t', 'name')}'} {
        public:
            $1 (${2:arguments});
            virtual ~$1 ();

        private:
            ${3:/* data */}

    };
# Function
snippet fun
    ${1:void} ${2:function_name}(${3}) {
        ${4:TODO}
    }
# Function Declaration
snippet fund
    ${1:void} ${2:function_name}(${3});${4}
=====
#结对符及特殊符合
=====
# 括号 bracket
snippet b
    (${1})${2}
# 方括号 square bracket
snippet sb
    [${1}]${2}
# 大括号 brace
snippet br
    {
        ${1}
    }${2}

```

```

# 单引号 single quotes
snippet sq
    '${1}'${2}
# 双引号 quotes
snippet q
    "${1}"${2}
# 中文括号 kuo hao
snippet kh
    (${1})${2}
# 中文双引号 yin hao
snippet yh
    "${1}" ${2}
# 指针间接访问 p->m
snippet .
    ->${1}${2}
# =====

```

另外，了解如下信息，有助于提高结对符内的字符编辑效率。以括号结对符为例，有六种常用的操作 `va()`、`vi()`、`da()`、`di()`、`ca()`、`ci()`，其中，`v` 是选中，`d` 是删除，`c` 是删除后插入，`a` 是包括结对符在内的整个文本对象，`i` 是结对符内部的文本对象（`inner`）。如，`va{选中包括结对符自身}` 在内的所有字符，`di[删除不包括结对符自身]` 之内的字符串；

第二类补全就真的智能了（通过 `<leader>tab` 激活）。实现智能补全的原理比较简单，先由后端分析工具将源码中所有函数、结构、成员、对象、变量、宏等等的名字、所在文件路径、定义、类型等信息（称之为标签信息）保存到一个独立文件（称之为 `tags` 数据库文件）中，再由 `vim` 和对应智能补全插件根据数据库信息快速匹配输入的字符，若找到匹配的则以列表形式显示之。目前比较知名的分析工具和补全插件有两个组合：传统的 `ctags` 与 `OmniCppComplete`、新生代的 `clang` 与 `clang complete`。由于 `ctags` 生成的 `tags` 数据库文件不仅可用于智能补全，还可支持其他插件实现函数、变量定义查找等功能（后面将介绍），所以我选用组合一。

- 软件：`ctags`
- 插件：`new-omni-completion`（内置）
- 操作：A、生成标签数据库文件。在你项目所在目录的顶层执行 `ctags`，该目录下会多出个 `tags` 文件；

B、在 `vim` 中引入标签数据库文件。在 `vim` 中执行命令

```
:set tags=/home/your_proj/tags
```

如果项目中要引入多个 tags 文件，可用 tags+= 的方式追加；

C、在 .vimrc 中增加如下配置信息：

```
"开启文件类型侦测
filetype on
"根据侦测到的不同类型加载对应的插件
filetype plugin on
"根据侦测到的不同类型采用不同的缩进格式
filetype indent on
"取消补全内容以分割子窗口形式出现，只显示补全列表
set completeopt=longest,menu
```

D、需要进行函数名、变量名、结构名、结构成员补全时输入 Ctrl+X

Ctrl+O，需要头文件名补全时输入 Ctrl+X Ctrl+I，需要文件路径补全时输入 Ctrl+X Ctrl+F。

实现 C 标准库函数智能补全

- 首先，获取 C 标准库头文件文件。安装的 GNU C 标准库源码文件（opensuse 可用如下命令：zypper install glibc-devel），安装成功后，在 /usr/include/ 目录中可见相关头文件；

- 然后，生成 C 标准库 tags：

```
cd /usr/include/
ctags --c-kinds=+l+x+p --fields=+S -f glibc.tags
```

- 最后，记着在 vim 中引入该 tags，在 .vimrc 中增加如下内容：

```
set tags+=/usr/include/glibc.tags
```

```
hardinfo.c+ (~/.downloads/hardinfo-0.5.l) - VIM
File Edit Tools Syntax Buffers Window DrChip C/C++ Help

[1:hardinfo.c]*+[6:util.c][8:blowfish.c][9:computer.c]
-MiniBufExplorer-
10 benchmark.c
11 benchmark.conf
12 benchmark.data
13 binreloc.c
14 blowfish.c
15 callbacks.c
16 computer.c
[File List]
- hardinfo.c (/root
- variable
  params
- function
  main
~
~
~
~
Tag_List hardinf
-- Omni completion (^O^N^P) match 3 of 44

39 if (!g_thread_supported())
40 g_thread_init(NULL);
41
42 /* parse all command line parameters */
43 parameters_init(&argc, &argv, &params);
44
45 scan[]
46 scan_arp( f void @@(gboolean
47 scan_battery( f void @@(gboolean
48 scan_bfsh( f void @@(gboolean
49 scan_boots( f void @@(gboolean
50 scan_boots_real( f @@(void) - arch/l
51 scan_connections( f void @@(gboolean
52 scan_cryptohash( f void @@(gboolean
53 scan_device_resources( f void @@(gboolean
54 scan_display( f void @@(gboolean
55 scan_dmi( f void @@(gboolean
56 scan_dns( f void @@(gboolean
57 scan_env_var( f void @@(gboolean
58 scan_fft( f void @@(gboolean
scan_fib( f void @@(gboolean
```

(C 库函数名补全)

实现 CPP 标准 (STL) 智能补全

以上是针对 C 语言的智能补全，对于 CPP，原理一样，同样需要 ctags 生成 CPP 类、模板等的 tags 文件，再在 vim 中引入生成的 tags，最后编码时由相应插件实时搜索 tags 中的类或模板，显示匹配项：

- 插件：OmniCppComplete
- 操作：对象后追加 “.” 或指向对象的指针后追加 “->” 时，自动显示成员列表；
- 注意：按如下步骤生成 CPP 标准库 tags。
 - 首先，获取 CPP 标准库源码文件。安装的 GNU C++ 标准库源码文件（opensuse 可用如下命令：zypper install libstdc++46-devel），安装成功后，在 /usr/include/c++/4.6 目录中可见所有源码文件；
 - 接着，执行 ctags 生成 CPP 标准库的 tags 文件：

```
cd /usr/include/c++/4.6
ctags -R --c++-kinds=+l+x+p --fields=+iaS --extra=+q
```

```
--language-force=c++ -f stdcpp.tags
```

- 然后，让 OmniCppComplete 识别标准库中各个标签。C++ 标准库源码文件中使用了 `_GLIBCXX_STD` 名字空间（gcc 使用的 STL 是这样，如果你使用其他版本的 STL，需要自行查找对应的名字空间名称），tags 文件里面的各个标签都嵌套在该名字空间下，所以，要让 OmniCppComplete 正确识别这些标签，必须显式告知 OmniCppComplete 相应的名字空间名称。在 .vimrc 中增加如下内容：

```
let OmniCpp_DefaultNamespaces = ["_GLIBCXX_STD"]
```

- 最后，在 vim 中引入该 tags 文件。在 .vimrc 中增加如下内容：

```
set tags+=/usr/include/c++/4.6/stdcpp.tags
```

顺便说下，stdcpp.tags 是 glibc.tags 的超集。即，若仅用 C 标准库则引入 glibc.tags，若同时使用 C 标准库和 CPP 标准库则引入 stdcpp.tags。

The screenshot shows a Vim editor window titled "business_hall_ping_ip.cpp + (~/.Desktop/tmp_proj) - VIM". The main editor area displays a C++ file with line numbers 65 to 90. The code includes a variable declaration, a typedef, and a function definition. A Tag List is visible on the left side of the editor, listing various symbols from the file. On the right side, a list of STL member functions is displayed, including `append`, `assign`, `at`, `begin`, `c_str`, `capacity`, `clear`, `compare`, `copy`, `data`, `empty`, `end`, `erase`, `find`, `find_first_not_of`, `find_first_of`, `find_last_not_of`, `find_last_of`, `get_allocator`, `insert`, `length`, `max_size`, `npos`, and `push_back`. The bottom status bar shows "-- Omni completion (^O^N^P) Back at original".

```
" Press <F1> 65
66     string str;
- business_hall 67     str.
| - variable    68         append( f + std::basic_st
| | kk          69         // assign( f + std::basic_st
| |             70         for at( f + std::basic_st
| - typedef     71         { begin( f + std::basic_st
| | businessha 72         c_str( f + std::basic_st
| |             73         capacity( f + std::basic_st
| - function    74         clear( f + std::basic_st
| | get_busin 75         compare( p + std::basic_st
| | get_busin 76         } copy( p + std::basic_st
| | net_secti 77         // data( f + std::basic_st
| | main       78         for empty( f + std::basic_st
| |           79         end( f + std::basic_st
| |           80         erase( f + std::basic_st
| |           81         find( p + std::basic_st
| |           82         // find_first_not_of( p + std::basic_st
| |           83         str find_first_of( f + std::basic_st
| |           84         int find_last_not_of( p + std::basic_st
| |           85         str find_last_of( f + std::basic_st
| |           86         int get_allocator( f + std::basic_st
| |           87         for insert( f + std::basic_st
| |           88         length( f + std::basic_st
| |           89         max_size( f + std::basic_st
| |           90         npos m + std::basic_st
__Tag_List__   business_hall_p push_back( f + std::basic_st
-- Omni completion (^O^N^P) Back at original
```

(STL 对象成员补全)

实现 linux 系统 API 智能补全

与上面实现 C 库函数、STL 智能补全类似，唯一需要注意 linux 系统 API 头文件中使用了 gcc 编译器扩展语法，必须告诉 ctags 忽略之，否则将生产错误的标签索引。

- 首先，获取 linux 系统 API 头文件。安装 linux 系统 API 头文件（opensuse 可用如下命令：`zypper install linux-glibc-devel`），安装成功后，在 `/usr/include/` 目录中可见相关头文件；
- 接着，执行 ctags 生成系统 API 的 tags 文件。由于 linux 自身采用 gcc 编译器，为提高内核执行效率，头文件中大量采用 gcc 扩展语法（或称之为编译器指示符），这在一定程度上影响 ctags 生成正确的标签索引，必须借由 ctags 的 `-I` 命令参数告知忽略某些标签，若有多个忽略字符串之间用逗号分割。比如，在文件 `unistd.h` 中几乎每个 API 声明中都会出现 “`__THROW`” 和 “`__nonnull`” 关键字，前者目的是告诉 gcc 这些函数不会抛异常，尽量多、尽量深地优化这些函数，后者目的告诉 gcc 凡是发现调用这些函数时第一个实参为 NULL 指针则将其视为语法错误，的确，使用这些扩展语法方便了我们编码，但却影响了 ctags 正常解析，这时可用 `-I __THROW,__nonnull` 命令行参数让 ctags 忽略这些语法扩展关键字：

```
cd /usr/include/  
ctags -R --c-kinds=+l+x+p --fields=+S -I __THROW,__nonnull  
-f sys.tags
```

- 最后，在 vim 中引入该 tags 文件。在 `.vimrc` 中增加如下内容：

```
set tags+=/usr/include/sys.tags
```

概要之，不论是 C 库函数、CPP STL、（甚至）boost、ACE 这些重量级开发库，还是 linux 系统 API 均可遵循“下载源码（至少包括头文件）-执行 ctags 生产标签索引文件”的流程实现智能补全，若有异常，唯有如下两种可能：一是源码中使用了名字空间，借助 OmniCppComplete 插件的 `OmniCpp_DefaultNamespaces` 配置项解决；一是源码中使用了编译器扩展语法，借助 ctags 的 `-I` 参数解决（上例仅列举了少量 gcc 扩展语法，此外还有 `__attribute_malloc__`、`__wur` 等等大量扩展语法，具体请参见 gcc 手册。以后，如果发现某个系统函数无法自动补全，十有八九是头文件中使用了 gcc 扩展语法，先找到该函数完整声明，再将其使用的扩展语法加入 `-I` 列表中，最后运行 ctags 重新生产新 tag 文件即可）。

ctags 或 OmniCppComplete 无法补全函数形参 (parameter)

——临时解决方案。编码时将形参复制到函数体内当作局部变量，编译前再将其注释掉（为防止遗忘导致编译时顺利通过，可用不存在的变量对其初始化。如，形参 string name 可在函数体内临时写为 string name = x，而并未定义变量 x，所以，如果在编译前忘记将这行注释掉，编译器会提示语法错误，从而避免了出现语义错误的可能）

——方案二。CTRL-X CTRL-I；

——方案三。考虑 clang 与 clang complete

在重载操作符函数体内，ctags 或 OmniCppComplete 智能补全失效

——临时解决方案。编码时将重载操作符函数名临时改为一般字符串，如，operator_pp 而非 operator++

ctags 或 OmniCppComplete 无法补全构造初始化的对象。

如，list<string> ls(8); // NOP

——解决方案。vector<int> vi = {16}; // YES

(待完善...) 实现 gnome 库函数智能补全

<http://developer.gnome.org/more>

前面提到过，智能补全是通过 tags 文件来实现的，如果代码中新增了函数或者调整了变量名，tags 文件无法自动更新，那么调整部分函数名或变量名肯定无法实现智能补全了，除非你手动再次执行 ctags -R . 命令。要是能自动更新 tags 文件就好了！哈哈，有求必应，开源世界就是好，隆重推出自动生成并实时更新 tags 文件的插件——indexer。

- 插件名：indexer
- 操作：必须先创建名为 .indexer_files 的配置文件且必须位于 \$HOME，指定要被 ctags 处理的文件类型及项目根目录，配置文件大致如下：

```
----- ~/.indexer_files -----  
[PROJECTS_PARENT filter="*.c *.h *.cpp"]  
/data/workplace/
```

这样，从 ~/workspace (及其子目录) 打开任何代码文件时，indexer 插件便对整个

目录及子目录生成 tags 文件，若代码文件有更新并保存时，indexer 插件自动更新项目的 tags 文件（indexer 插件生成的 tags 文件并未放在你工程目录下，而是在 ~/.indexer_files_tags 目录下，并以工程目录名命名 tags 文件）

- 注意：要使用该插件必须得让 ctags 软件达到 5.8.1 版本，ctags 官网上并无该版本，可在 <http://dfrank.ru/ctags581/en.html> 下载，安装后用 ctags --version 确认下版本是否正确。

【函数调用参考】

有过 Win32 SDK 开发经验的朋友对 MSDN 或多或少有些迷恋吧，对于多达 7、8 个参数的 API，如果没有一套函数功能描述、参数讲解、返回值说明的文档，那么软件开发将是人间炼狱。别急，vim 也能做到。这里说的系统函数，包括操作系统自身提供的 API 和 C/CPP 语言库函数（方法、STL）。

- 插件名：man.vim（内置）
- 操作：先在 vim 中启动该插件：source \$VIMRUNTIME/ftplugin/man.vim（可以加入 .vimrc 中自动启动该插件），需要查看系统函数参考时输入：Man sys_api 即可在新建分割子窗口中查看到 sys_api() 函数参考信息（你没看错，vim 中输入的是 Man，shell 中输入的是 man）；
- 注意：要使用该功能，系统中必须先安装对应 man。
 - ➔ linux 风格的 C 标准库函数和系统库函数：安装 man-pages；
 - ➔ posix 风格的 C 标准库函数和系统库函数：安装 man-pages-posix。由于 linux 实现时完全遵循 posix 规范，所以两者基本差不多，posix 在 man 列表中带有 p 后缀，如，scanf (3p) 帮助手册；
 - ➔ C++ 标准类库：没有现成的 man 安装文件，只能手工下载后拷贝到系统 man 目录中（至少在 openSUSE 下是这样）。先到 <ftp://gcc.gnu.org/pub/gcc/libstdc++/doxygen/> 下载 libstdc++-api-X.X.X.man.tar.bz2，解压后将 libstdc++-api-X.X.X.man/man3/ 目录内的文件拷贝至 /usr/share/man/man3/ 即可。另外，查看时一定要加上 std 前缀，如，:Man std::vector。

通过以上设置，已经可以在 vim 中直接查看到 C 和 CPP 标准库相关函数和类的调用说明，但每次输入 “:Man std::vector” 挺麻烦的，如果能实现输入快捷键后 vim 自动显示光标所在单词的 man 信息该多好啊！前面说过，vim 给你足够发挥的空间，只要你能想到她就能做到。通过如下几步可实现输入 “;man” 快捷键后快速查看当前光标所在单词的 man 信息：

1. 我们编码时通常都是先声明使用 std 名字空间，在使用某个标准库中的类时前不会添加 std::前缀，所以 vim 取到的当前光标所在单词中也不会含有 std::前缀，但，libstdc++ 中各 man 文件名都有 std 前缀，所以只有将所有文件的 std::前缀去掉才能让:Man (man 命令也是一样) 找到正确的 man 文件。你可以这样实现：

```
#先进入 libstdc++-api-X.X.X.man/man3/目录
cd libstdc++-api-X.X.X.man/man3/
#再执行批量重命名以取消 libstdc++ 的所有 man 文件 std::前缀
rename "std::" "" std:.*
#最后将更名后的 man 文件拷贝至系统 man3 目录
mv * /usr/share/man/man3/
```

顺便说下，很多人以为 rename 命令只是 mv 命令的简单封装，非也，在重命名方面，rename 太专业了，远非 mv 可触及滴，就拿上例来说，mv 必须结合 sed 才能达到这样的效果；

2. 映射快捷键，让 vim 自动提取光标当前所在单词并传递给 vim 自带的:Man 命令。将如下配置信息添加进.vimrc 即可：

```
" 启用:Man 命令查看各类 man 信息
source $VIMRUNTIME/ftplugin/man.vim
" 定义:Man 命令查看各类 man 信息的快捷键
nmap <Leader>man :Man 3 <cword><CR>
```

如下图所示：

The screenshot shows a Vim editor window titled "hardinfo.c (~/.downloads/hardinfo-0.5.l) - VIM". The menu bar includes File, Edit, Tools, Syntax, Buffers, Window, DrChip, C/C++, and Help. The toolbar contains various icons for file operations and editing. The main editor area is split into two panes. The left pane shows a file list for "MiniBufExplorer-" with files: 32 report.c, 33 sha1.c, 34 shell.c, 35 socket.c, 36 stock.c, 37 syncmanager.c, and 38 tags. Below this is a "[File List]" section with a prompt "Press <F1> to d". The right pane shows the manual page for "FOPEN(3) Linux Programmer's Manual FOPEN(3)". The manual page includes sections for NAME, SYNOPSIS, and a code snippet. The NAME section describes "fopen, fdopen, freopen - stream open functions". The SYNOPSIS section shows the header file "#include <stdio.h>" and the function signature "FILE *fopen(const char *path, const char *mode);". The code snippet shows a C program with lines 39 to 49, including a conditional compilation block for threads and a main function that parses command-line arguments and prints version information. The bottom of the window shows a "Tag_List" section with "hardinfo.c [+]" and a command prompt ":Man fopen".

```
hardinfo.c (~/.downloads/hardinfo-0.5.l) - VIM
File Edit Tools Syntax Buffers Window DrChip C/C++ Help

[1:hardinfo.c]*+[6:util.c][8:blowfish.c][9:computer.c][10:network.c][11:syncmana
ger.c]
-MiniBufExplorer-
32 report.c
33 sha1.c
34 shell.c
35 socket.c
36 stock.c
37 syncmanager.c
38 tags
[File List]
" Press <F1> to d
- hardinfo.c (/root
|- variable
||   params
|- function
||   main
~
~
~
~
~
~
Tag_List
:Man fopen
```

FOPEN(3) Linux Programmer's Manual FOPEN(3)

NAME
fopen, fdopen, freopen - stream open functions

SYNOPSIS
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);

[Scratch]

```
39  if (!g_thread_supported())
40    g_thread_init(NULL);
41
42  /* parse all command line parameters */
43  parameters_init(&argc, &argv, &params);
44
45  fopen();
46
47  /* show version information and quit */
48  if (params.show_version) {
49    g_print("HardInfo version " VERSION "\n");
```

(C 库函数调用参考)

```
std::vector(3)                                std::vector(3)
std::vector(3)

NAME
    std::vector -

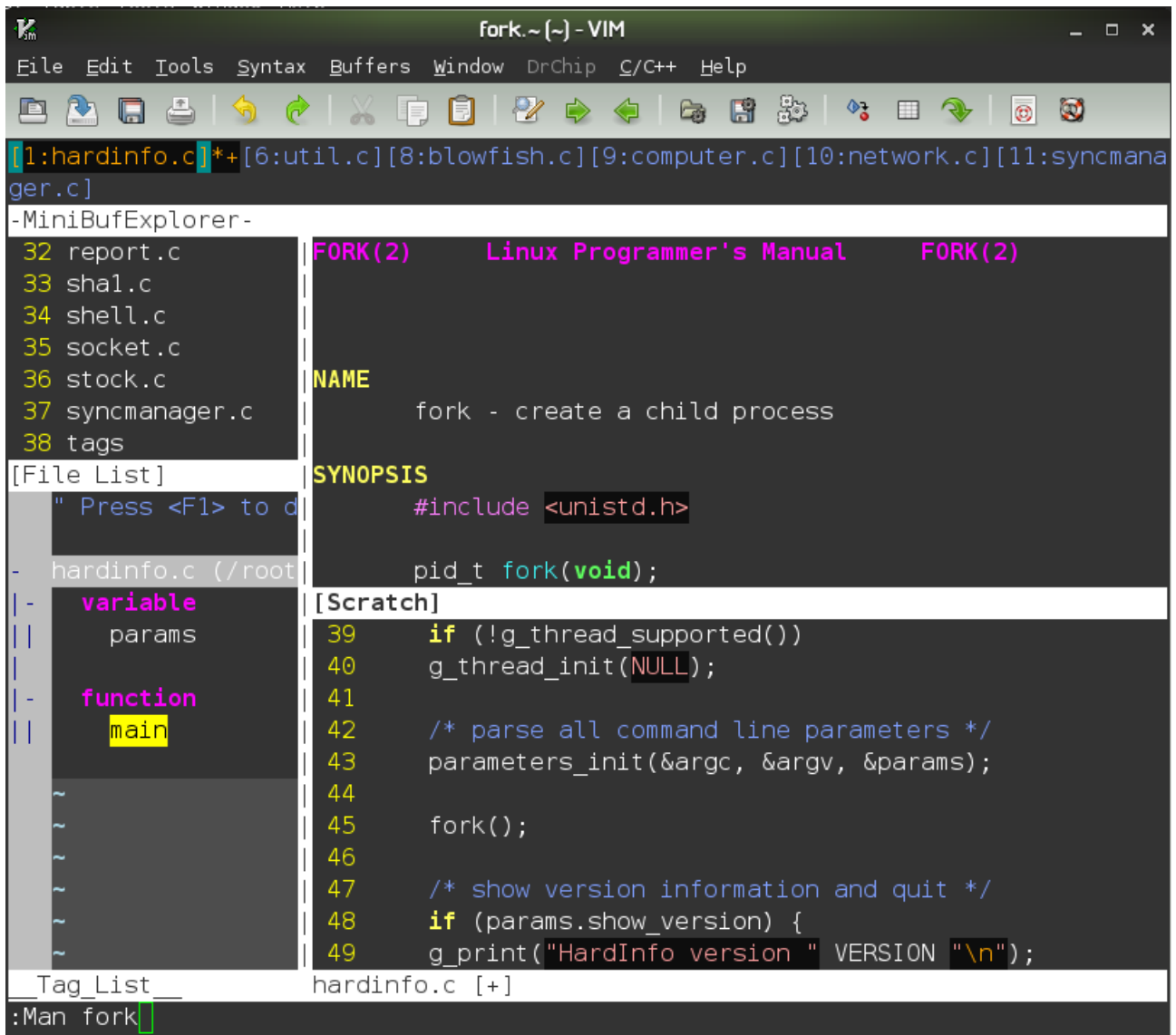
    A standard container which offers fixed time access to
    individual elements in any order.

SYNOPSIS
    Inherits std::_Vector_base< _Tp, _Alloc >.

Public Types
    typedef _Alloc allocator_type
    typedef __gnu_cxx::__normal_iterator< const_pointer, vector >
    const_iterator
    typedef _Tp_alloc_type::const_pointer const_pointer
    typedef _Tp_alloc_type::const_reference const_reference

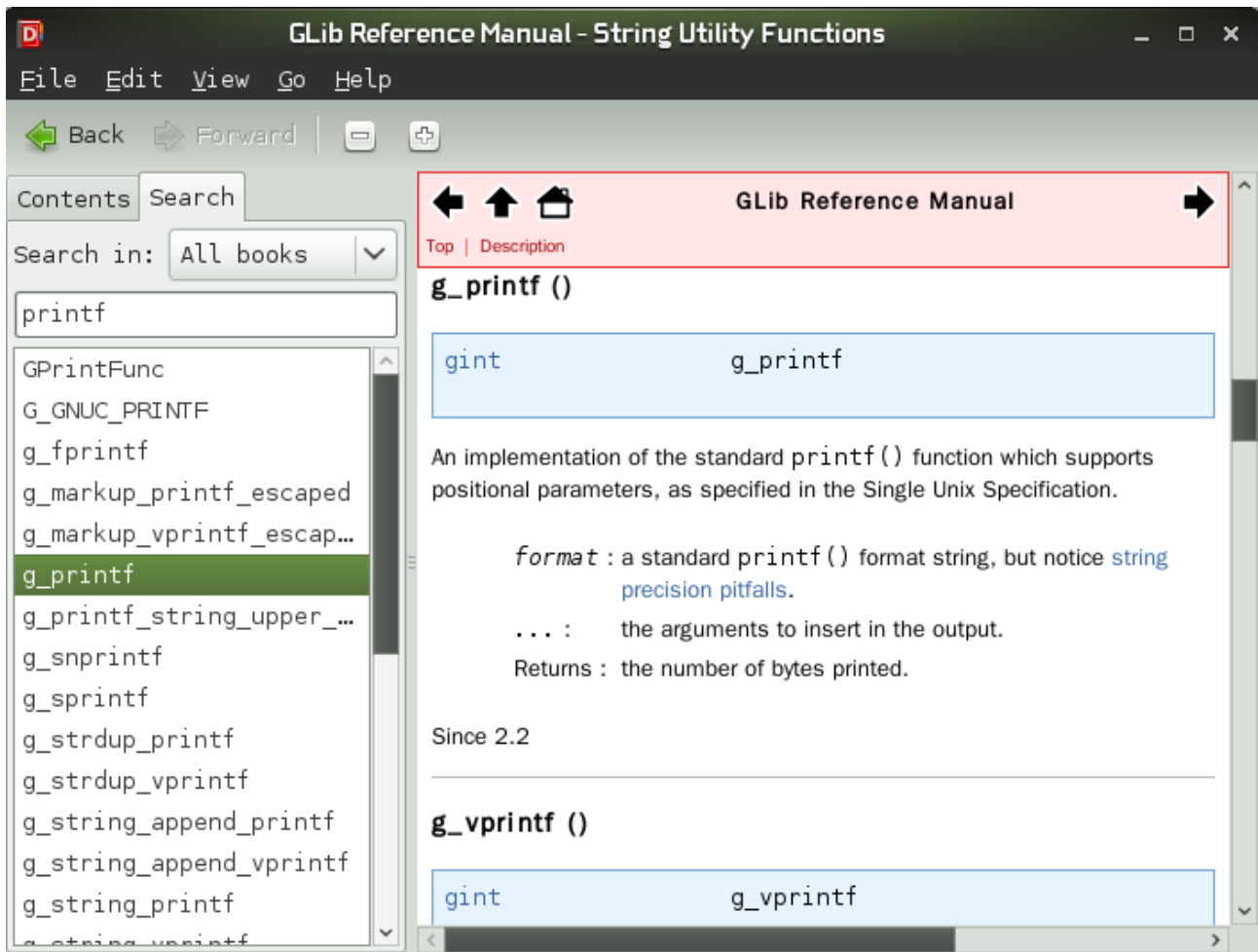
[Scratch]                                     14,0-1      Top
252     while (!file_allot_net_section.eof ())
253     {
254         char    buffer[512];
255         file_allot_net_section.getline (buffer, 256);
256         string  str(buffer);
257         if (" " == str)
258             break;
259         city_allot_net_section.push_back (str);
260     }
business_hall_ping_ip.cpp                     257,16-22     78%
:Man std::vector
```

(CPP 标准库参考)



(系统函数调用参考)

以上介绍了如何在 VIM 中集成查阅各类函数参考，如果你更喜欢独立的 HTML 参考手册，可到 <http://en.cppreference.com/w/Cppreference:Archives> 下载或到 <http://www.cplusplus.com/reference/> 在线访问（优选后者）；此外，如果你从事 gnome 开发，还可以安装独立软件 devhelp，这是 GTK 版的 MSDN。如下图所示：



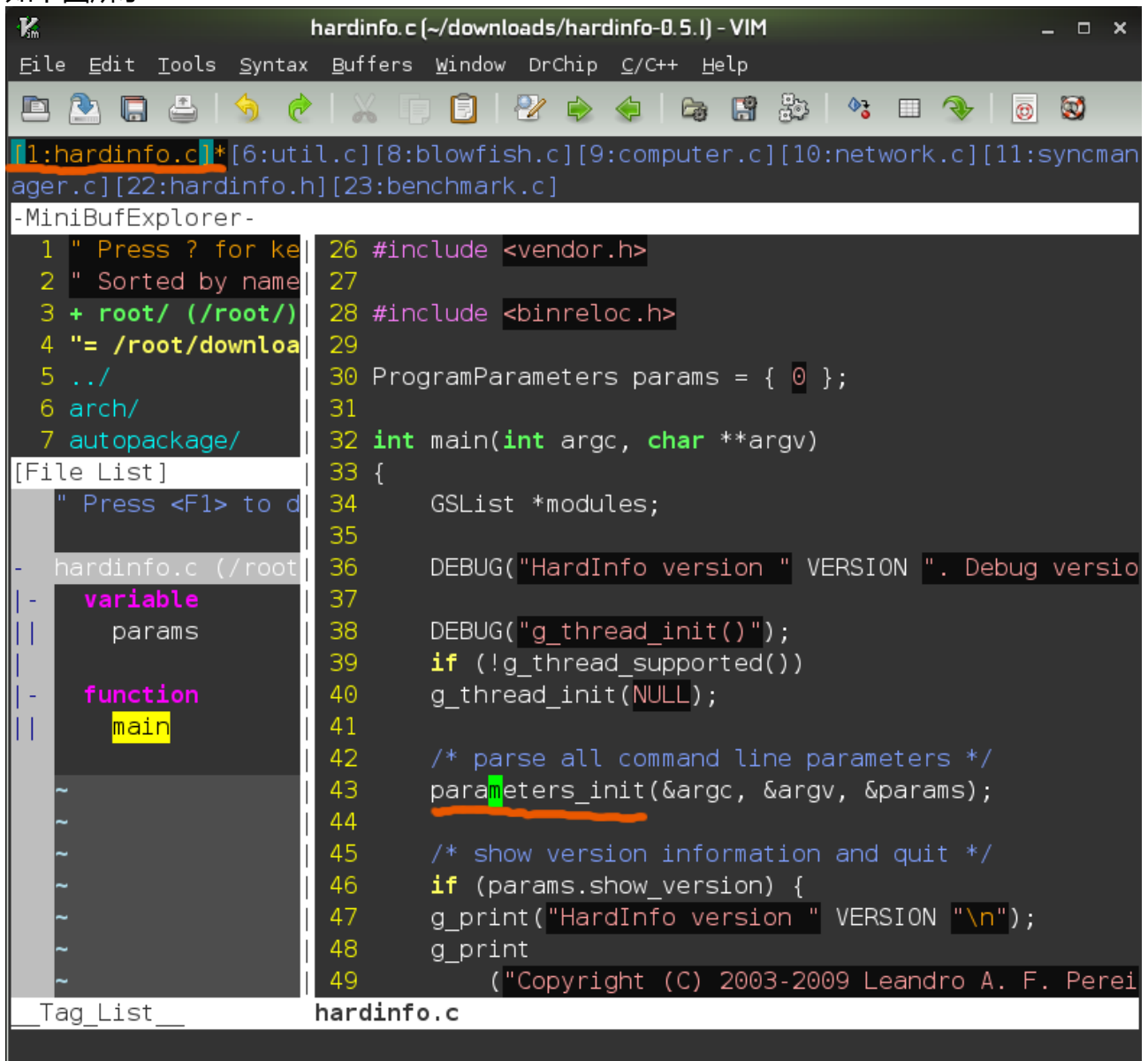
(gnome 函数调用参考)

【快速查看定义】

上面说了如何查看系统函数调用，自己写的函数又如何快速查看其函数定义了？单代码文件的项目倒是不麻烦，无非上下拖动下滚动条或者多按几次 j、k 键而已，对于动辄十多个代码文件的一般项目来说，准确、快速查找到函数定义对于提高开发效率非常有帮助。下面介绍如何快速跳转到函数定义、变量定义、结构定义、成员定义处。

快速跳转到函数定义处，也是通过标签来实现的，所以必须得有 tags 文件支持，如果你还没实现前面“智能补全”部分的功能，那请倒回去看看，成功后再继续这部分内容，如果已经实现，你可以随便找个自定义函数（注，一定要是自定义函数，因为 ctags 未对系统函数生成标签，对于系统函数，我一般只关注其功能、参数、返回值等信息，不会关注起实现。前者通过“系统函数调用参考”部分已经实现，后者如果你的确需要，可将“ctags -R .”替换成“ctags -R --fields=+lS /usr/include”，“/usr/include”为系统函数头文件和实现文件所在目录），把光标移到上面，输入 CTRL-] 或者 g]，呵呵，是不是奇迹发生了呀。

比如，在 hardinfo.c 文件中调用函数 parameters_init()，将光标移到该函数上，如下图所示：

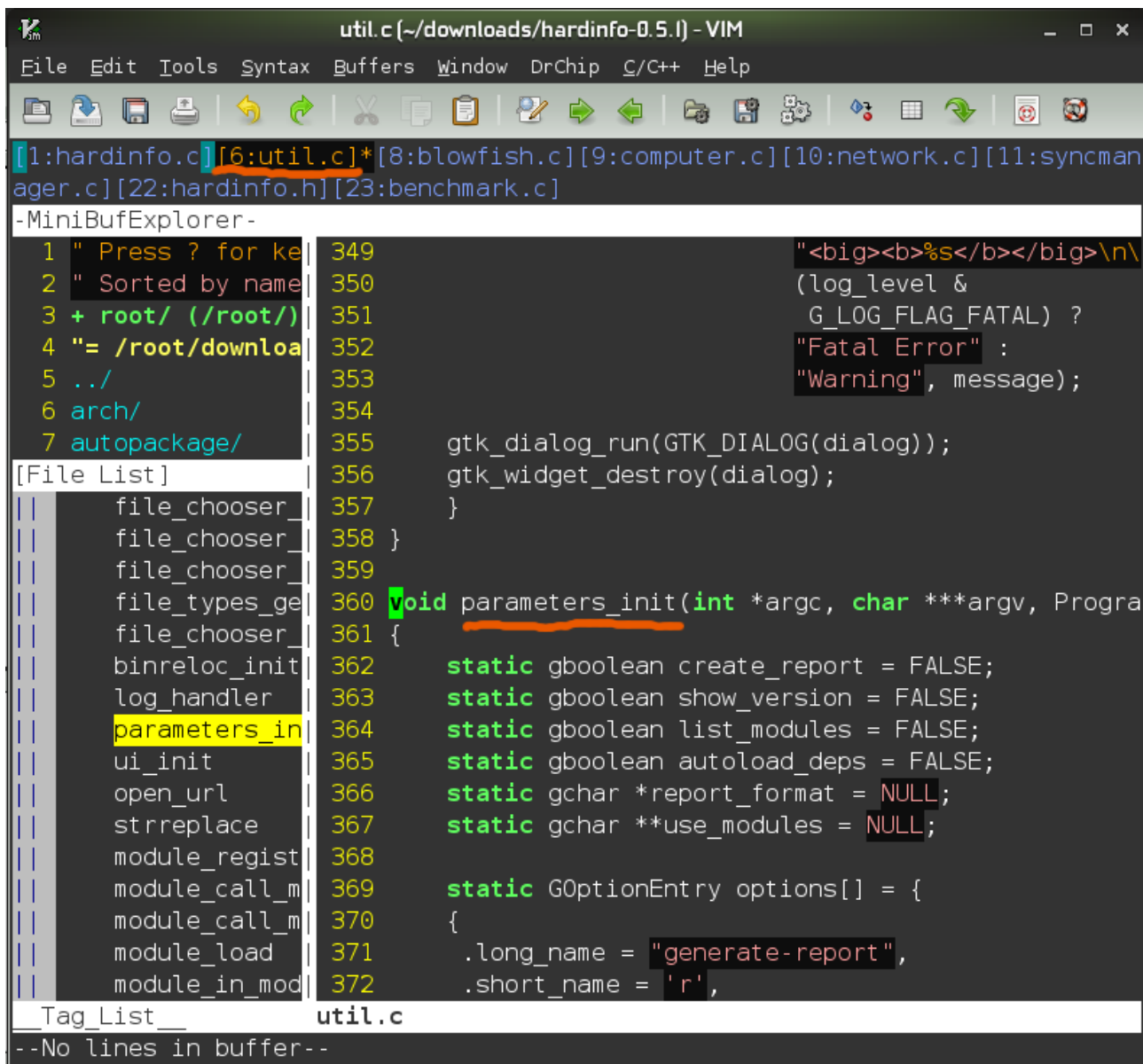


```
hardinfo.c (~/.downloads/hardinfo-0.5.l) - VIM
File Edit Tools Syntax Buffers Window DrChip C/C++ Help

[1:hardinfo.c]*[6:util.c][8:blowfish.c][9:computer.c][10:network.c][11:syncman
ager.c][22:hardinfo.h][23:benchmark.c]
-MiniBufExplorer-
1 " Press ? for ke 26 #include <vendor.h>
2 " Sorted by name 27
3 + root/ (/root/) 28 #include <binreloc.h>
4 "= /root/downloa 29
5 ../ 30 ProgramParameters params = { 0 };
6 arch/ 31
7 autopackage/ 32 int main(int argc, char **argv)
[File List] 33 {
" Press <F1> to d 34     GSList *modules;
35
36     DEBUG("HardInfo version " VERSION ". Debug versio
37
38     DEBUG("g_thread_init()");
39     if (!g_thread_supported())
40     g_thread_init(NULL);
41
42     /* parse all command line parameters */
43     parameters_init(&argc, &argv, &params);
44
45     /* show version information and quit */
46     if (params.show_version) {
47     g_print("HardInfo version " VERSION "\n");
48     g_print
49     ("Copyright (C) 2003-2009 Leandro A. F. Perei
Tag_List__ hardinfo.c
```

(准备跳转到函数定义处)

输入 CTRL-] 后，通过 tags 文件，vim 找到 parameters_init() 定义在 util.c 文件的 360 行，并自动定位到该文件的正确行数上，如下图所示：



```
util.c (~/.downloads/hardinfo-0.5.l) - VIM
File Edit Tools Syntax Buffers Window DrChip C/C++ Help

[1:hardinfo.c][6:util.c]*[8:blowfish.c][9:computer.c][10:network.c][11:syncman
ager.c][22:hardinfo.h][23:benchmark.c]
-MiniBufExplorer-
1 " Press ? for ke 349 "<big><b>%s</b></big>\n\
2 " Sorted by name 350 (log_level &
3 + root/ (/root/) 351 G_LOG_FLAG_FATAL) ?
4 "= /root/downloa 352 "Fatal Error" :
5 ../ 353 "Warning", message);
6 arch/ 354
7 autopackage/ 355 gtk_dialog_run(GTK_DIALOG(dialog));
[File List] 356 gtk_widget_destroy(dialog);
|| file_chooser 357 }
|| file_chooser_ 358 }
|| file_chooser_ 359 }
|| file_types_ge 360 void parameters_init(int *argc, char ***argv, Progra
|| file_chooser_ 361 {
|| binreloc_init 362 static gboolean create_report = FALSE;
|| log_handler 363 static gboolean show_version = FALSE;
|| parameters_in 364 static gboolean list_modules = FALSE;
|| ui_init 365 static gboolean autoload_deps = FALSE;
|| open_url 366 static gchar *report_format = NULL;
|| strreplace 367 static gchar **use_modules = NULL;
|| module_regist 368
|| module_call_m 369 static GOptionEntry options[] = {
|| module_call_m 370 {
|| module_load 371 .long_name = "generate-report",
|| module_in_mod 372 .short_name = 'r',
__Tag_List__ util.c
--No lines in buffer--
```

(正确跳转到函数定义处)

OK，上面演示了如何快速跳转到函数定义处，变量、结构、成员也是类似的。另外，**CTRL】** 跳转到定义处后，如果此时你还想再跳回先前位置，按 **CTRL-O**，前进按 **CTRL-I**。

此外，有时我们在阅读代码时，希望有个窗口能将当前代码文件中所有函数名、变量名、结构名、类成员等以列表形式罗列出来，以便有针对性的分析代码。呵呵，可以滴~。

- 插件名：tagbar 插件
- 操作：输入:TagbarToggle 即可调出标签列表子窗口，切换到不同代码文件时，列表会自动更新，s 对列表进行重新排序；
- 注意：请在.vimrc 中增加如下配置信息：

"定义快捷键的前缀，即<Leader>

```

let mapleader=";"
"设置显示 / 隐藏标签列表子窗口的快捷键。速记：tag list
nnoremap <Leader>t1 :TagbarToggle<CR>
"设置 tagbar 子窗口的位置出现在主编辑区的左边
let tagbar_left=1
"设置标签子窗口的宽度
let tagbar_width=20

```

针对类成员，tagbar 通过+、-、#三个符合依次表示公有成员、私有成员、被保护成员，很直观吧。

如下图左侧窗口所示：

```

business_hall_ping_ip.cpp + (~/Desktop/tmp_proj) - VIM
▼ typedefs
  businesshall_allo
▼ a : class
  +a()
  +~a()
  #GetName(bool b0p
  -_init(int timeou
  -m_ret
▼ b : class
  -b
  -bb()
▼ __anon_ktop_tmp_pr
  +businesshall_name
  +businesshall_allo
▼ functions
  get_businesshall_
  get_businesshall_
  net_section_to_i
  main(int argc, c
~
~
~
31 } businesshall_allot_ip;
32
33 class a
34 {
35 //pubilc:
36 ////    int a;
37 //  int aa();
38 public:
39     a();
40     virtual ~a();
41
42 protected:
43     int GetName(bool b0pen);
44
45 private:
46     int _init(int timeout);
47
48 private:
49     int m_ret;
50
51 };
52 class b : a
53 {
54     int b;
55     int bb();
56 };
<ss_hall_ping_ip.cpp <siness_hall_ping_ip.cpp [+] 43,5 10%

```

(标签列表)

【语法高亮】

现在已是千禧年后的十年了，早已告别上世纪六、七十年代黑底白字的时代，如果编码时没有语法高亮，肯定会让你的代码失去活力，即使在字符模式下编程（感谢伟大的 fbterm），我也会开启语法高亮功能。

vim 自身就支持语法高亮，只须打开相关设置即可。请将如下配置信息添加到.vimrc：

```
"打开语法高亮
set syntax enable
"允许按指定主题进行语法高亮，而非默认高亮主题
set syntax on
"指定配色方案
colorscheme evening
```

我选用内置的 evening 配色方案，灰底、白字、黄色关键字、蓝色注释.....，效果还不错，比较符合我的审美观。vim 内置了 10 多种配色方案供你选择，GUI 下，可以通过菜单（Edit -> Color Scheme）试用不同方案，字符模式下，需要你手工调整配置信息，再重启 vim 查看效果（推荐 csExplorer.vim 插件，可在字符模式下不用重启即可查看效果）。如果都不满意，可以去

<http://vimcolorschemetest.googlecode.com/svn/html/index-c.html> 慢慢选，喜欢的下载到 ~/.vim/colors 即可。

vim 默认对 .c 和 .cpp 文件启用 C 语言语法高亮，但对 C++ 语言的特有语法（如，STL）支持较差，为增强对 STL 语法高亮，须先下载对应的配色文件 stl.vim（http://www.vim.org/scripts/script.php?script_id=2224）后拷贝至 ~/.vim/after/syntax/cpp/ 目录重启 vim 即可。该配色文件仅包含了部分常用关键字，可按自己需要编辑该文件进行完善。如，随着 C++11 的发布，新增了大量容器和泛型算法，要想高亮显示这些新增元素，可在该文件基础上进行如下调整：

- 在以 "syn keyword cppSTL" 开头的那行行尾追加如下函数名（成员函数、泛型算法）：
cbegin cend crbegin crend all_of any_of none_of find_if_not
copy_if copy_n move move_backward random_shuffle is_partitioned
partition_point is_sorted is_sorted_until is_heap_until minmax
minmax_element is_permutation
- 在以 "syn keyword cppSTLtype" 开头的那行行尾追加如下容器名：
array
forward_list unordered_set unordered_multiset unordered_map

unordered_multimap

又如，要让 IO 操纵符高亮，可在“syn keyword cppSTL”行尾追加如下操纵符：

boolalpha showbase showpoint showpos skipws unitbuf uppercase noboolalpha
noshowbase noshowpoint noshowpos noskipws nounitbuf nouppercase dec hex
oct fixed scientific internal left right ws endl ends flush setiosflags
resetiosflags setbase setfill setprecision setw。

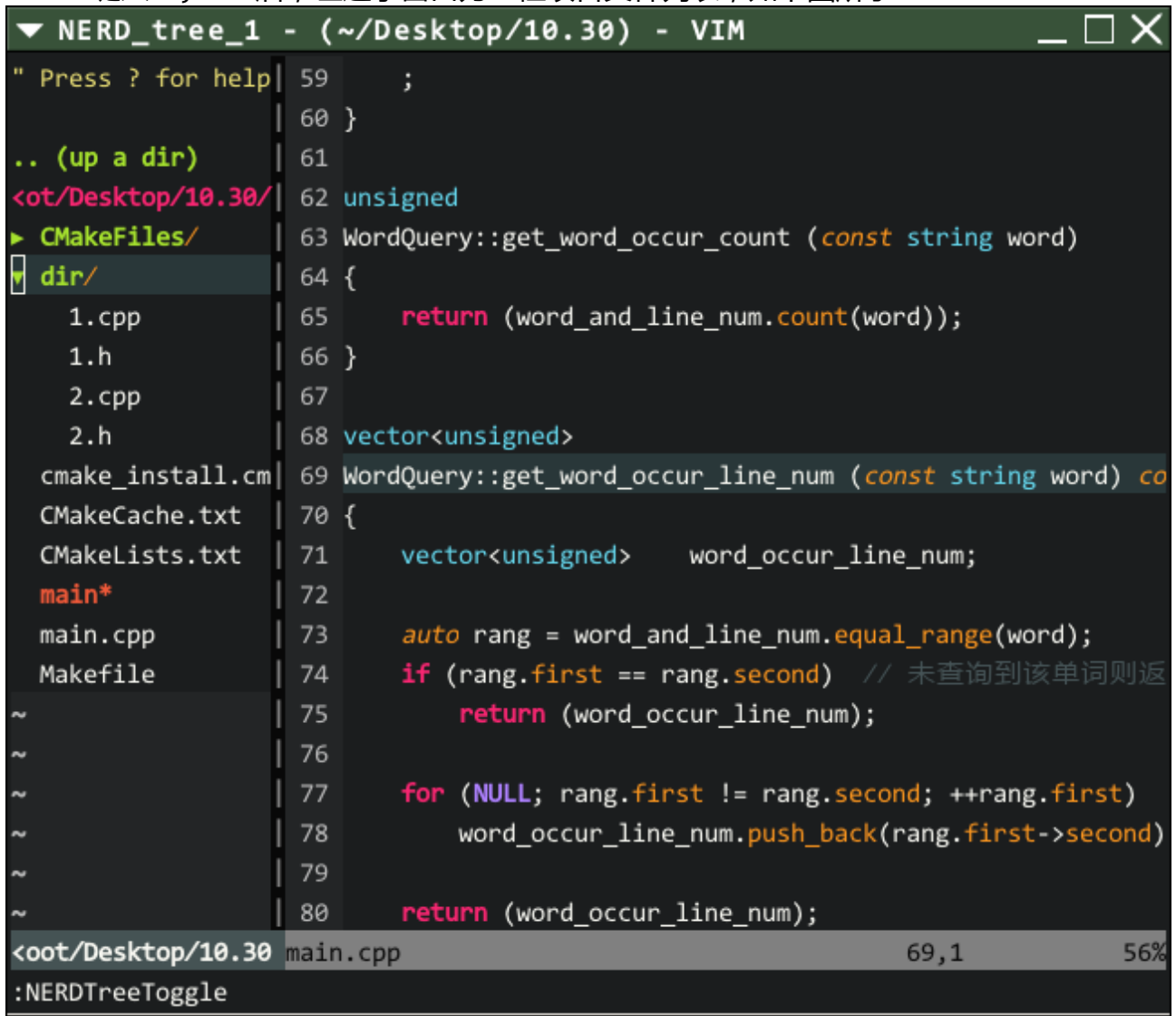
如下图所示，STL 中的 string、vector 等关键字都已高亮显示：



```
75
76 // 将网段转换为具体IP。
77 // 网段规则：1、不同网段用“;”分割，最后一个网段不能追加“;”；2、同网段>
78 //          后续IP只写最后一个字段，最后一个字段若连续的可用“-”缩写，
79 // 算法描述：先将同网段中连续IP扩展为独立IP，再完善所有独立IP的前3个字
80 vector<string>
81 net_section_to_ip (string net_section)
82 {
83     vector<string> all_ip;
84
85     // 扩展“-”缩写的连续IP
86     for ( string::size_type i = 0;
87          string::npos != (i = net_section.find("-", i));
88          ++i )
89     {
90         vector<int> ip_digit; // 存放数字型的连续IP，不包括“-”前后的
91         int before_ip_len = 0, after_ip_len = 0;
92
93
94
95         // 检查“-”前的IP位数，最多3位
96         for (int j = 1; j <= 3; ++j)
97         {
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
261
```

```
"使用 NERDTree 插件查看工程文件。设置快捷键，速记法：file list
nmap <Leader>fl :NERDTreeToggle<CR>
"设置 NERDTree 子窗口宽度
let NERDTreeWinSize=23
```

键入“;fl”后，左边子窗口为工程项目文件列表，如下图所示：



```
▼ NERD_tree_1 - (~/Desktop/10.30) - VIM
" Press ? for help | 59      ;
                   | 60  }
.. (up a dir)      | 61
<ot/Desktop/10.30/| 62 unsigned
► CMakeFiles/      | 63 WordQuery::get_word_occur_count (const string word)
▼ dir/             | 64 {
  1.cpp            | 65     return (word_and_line_num.count(word));
  1.h              | 66 }
  2.cpp            | 67
  2.h              | 68 vector<unsigned>
cmake_install.cm  | 69 WordQuery::get_word_occur_line_num (const string word) co
CMakeCache.txt    | 70 {
CMakeLists.txt    | 71     vector<unsigned>    word_occur_line_num;
main*             | 72
main.cpp          | 73     auto rang = word_and_line_num.equal_range(word);
Makefile          | 74     if (rang.first == rang.second) // 未查询到该单词则返
~                | 75     return (word_occur_line_num);
~                | 76
~                | 77     for (NULL; rang.first != rang.second; ++rang.first)
~                | 78         word_occur_line_num.push_back(rang.first->second)
~                | 79
~                | 80     return (word_occur_line_num);
<oot/Desktop/10.30 main.cpp 69,1 56%
:NERDTreeToggle
```

(项目文件列表)

【多文档编辑】

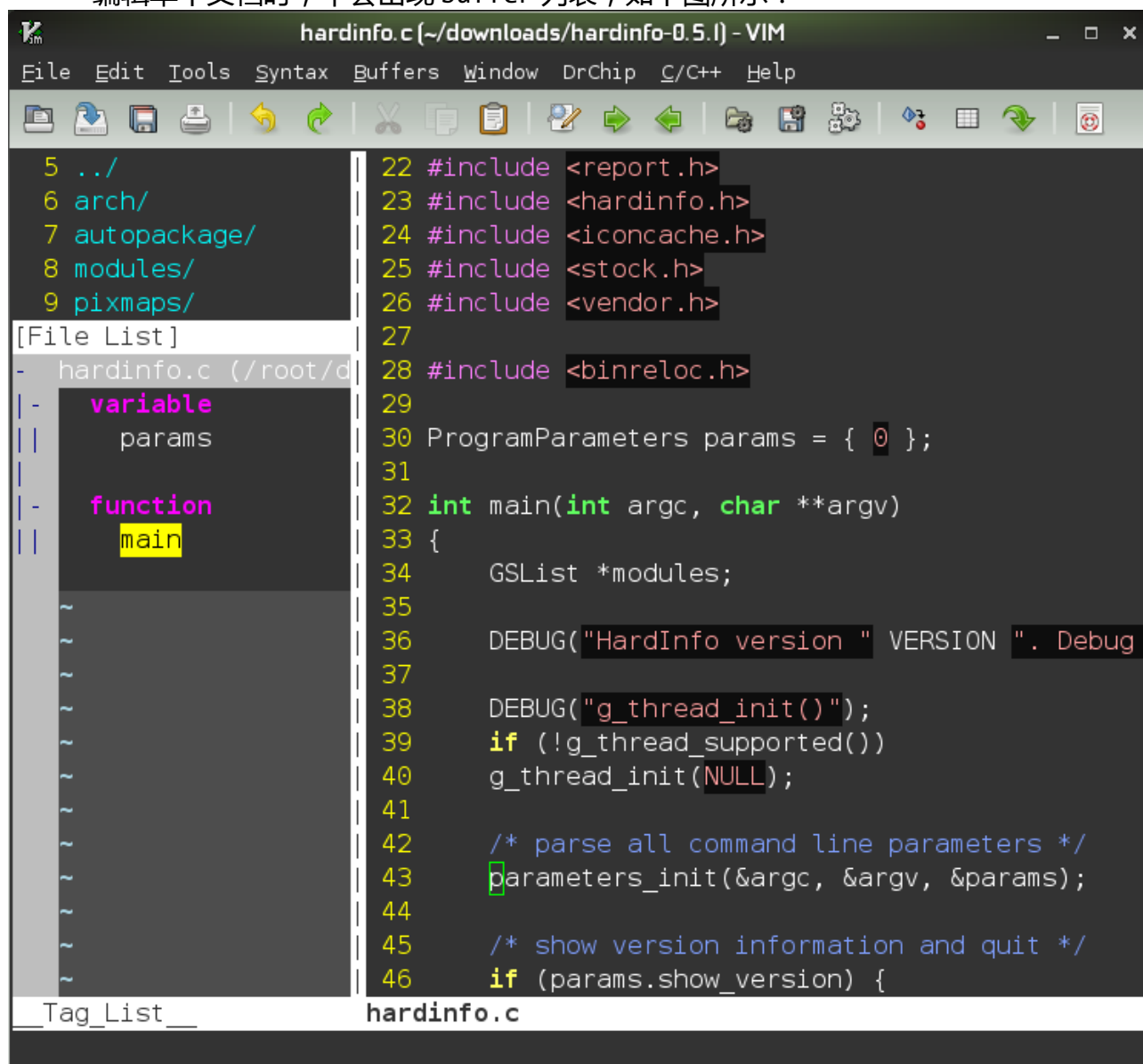
一个 EXCEL 文档可以有多个 SHEET，你可以在不同 SHEET 间来回切换，同样，编程时也需要类似功能，即，同时打开多个文件，可以自由自在地在不同代码文件间游历。这种需求，vim 是通过 buffer 来实现的。每打开一个文件 vim 就对应创建一个 buffer，多个文

件就有多个 buffer。

- 插件名：MiniBufExplorer
- 操作：打开多个文档时，vim 在窗口顶部自动创建 buffer 列表窗口。光标在任何位置时，CTRL-TAB 正向遍历 buffer，CTRL-SHIFT-TAB 逆向遍历；光标在 MiniBufExplorer 窗口内，输入 d 删除光标所在的 buffer
- 注意：将如下信息加入.vimrc 中：

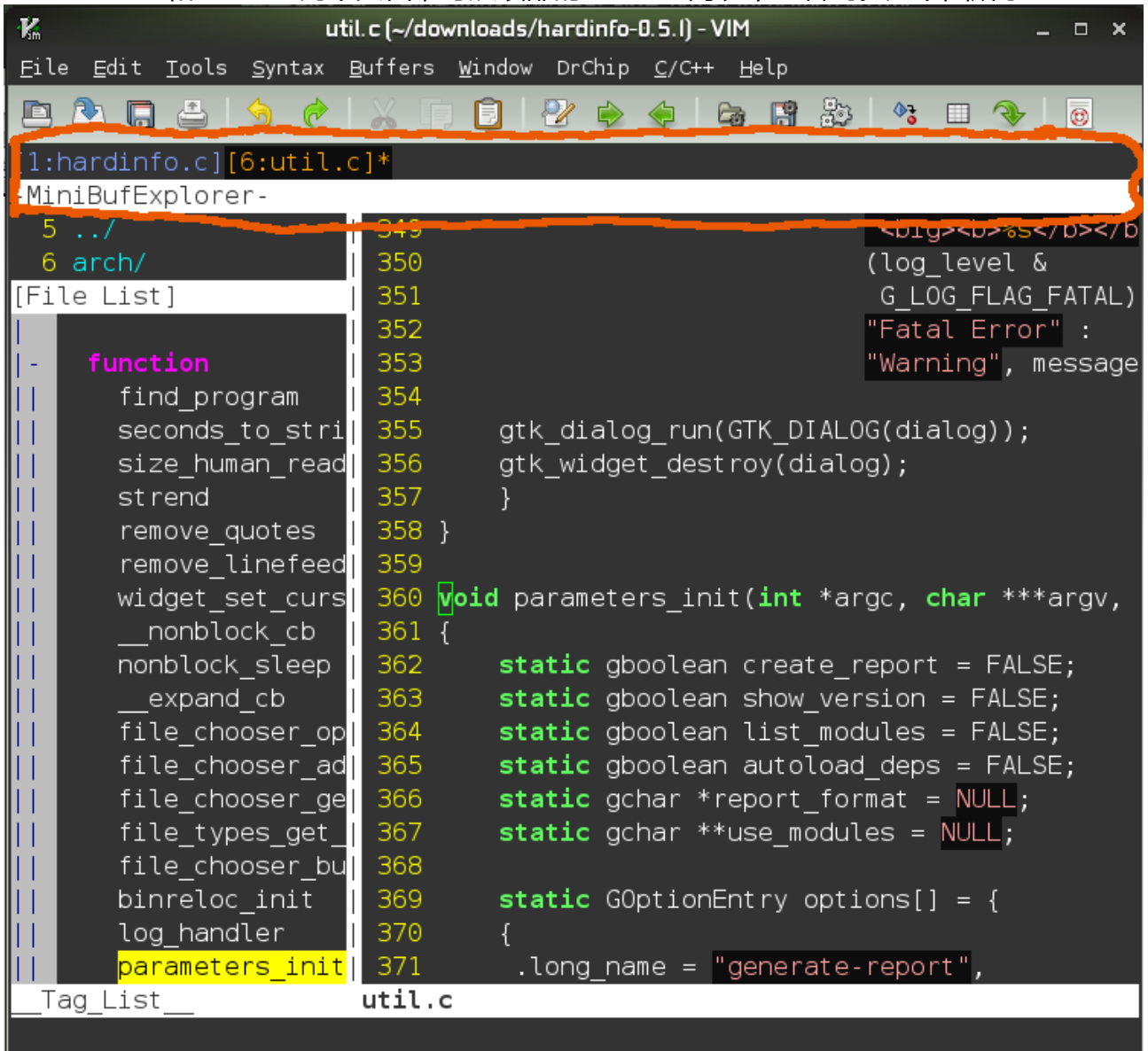
```
"允许光标在任何位置时用 CTRL-TAB 遍历 buffer
let g:miniBufExplMapCTabSwitchBufs=1
```

编辑单个文档时，不会出现 buffer 列表，如下图所示：



(编辑单个文档)

当前编辑的是 hardinfo.c 文件，该文件中有调用 parameters_init() 函数，但该函数定义在 util.c 文件中，当我在 parameters_init() 上输入 CTRL-] 后，vim 新建 util.c 文件的 buffer 并自动定位到 parameters_init()，这时，vim 同时在编辑 hardinfo.c 和 util.c 两个文档，可从顶部的 buffer 列表中查看到。如下图所示：



(同时编辑多个文档)

【代码折叠】

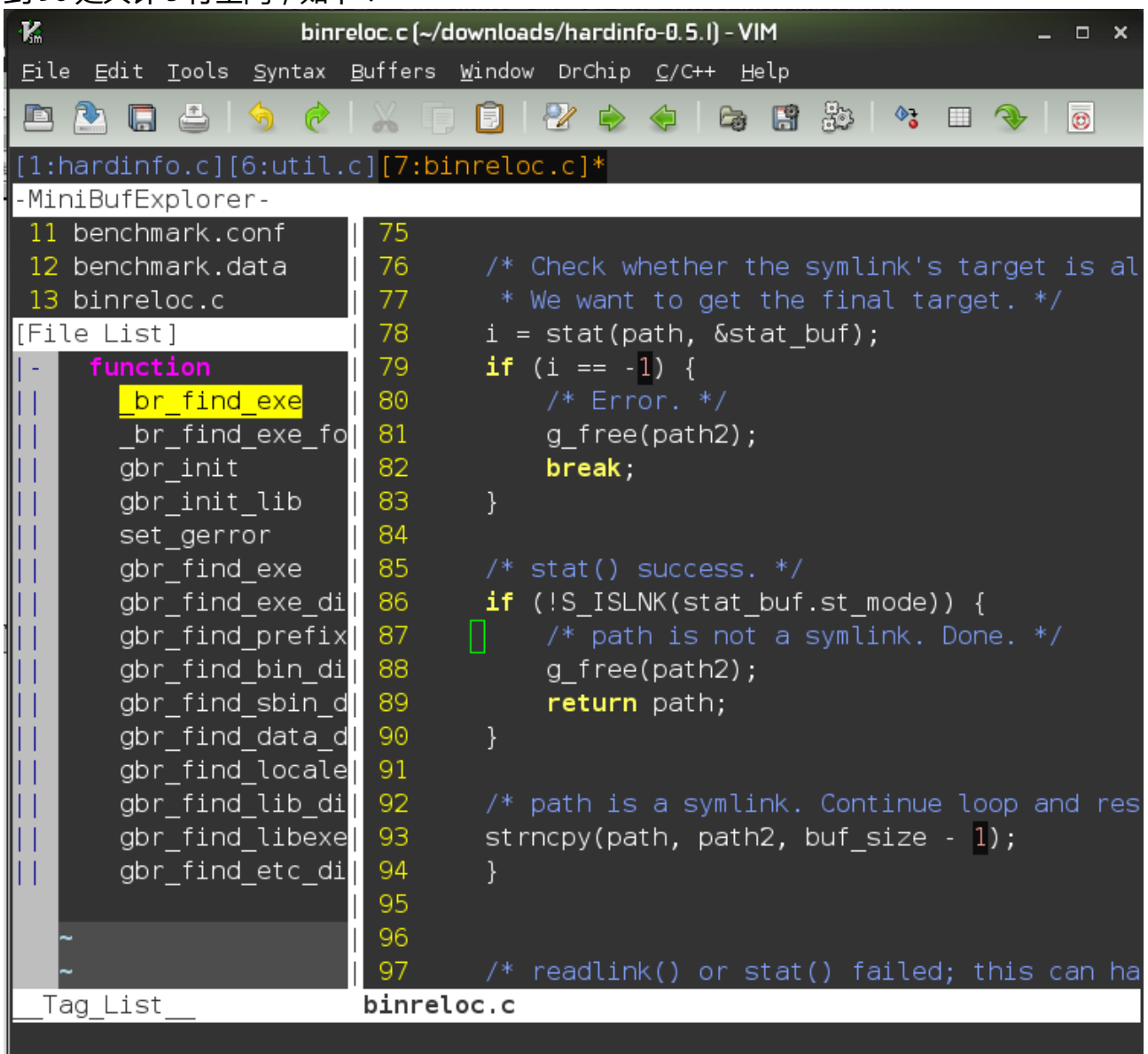
有时为了去除干扰，集中精力在某部分代码片段上，我会把不关注部分代码折叠起来。vim 自身支持多种折叠，包括：手动建立折叠 (manual)、相同缩进距离的行构成折叠 (indent)、'foldexpr' 给出每行的折叠 (expr)、标志用于指定折叠 (marker)、语法高亮项目指定折叠 (syntax)、没有改变的文本构成折叠 (diff)。用于编程时的折

叠当然就选“语法高亮项目指定折叠 (syntax)”啦。

- 操作：za，打开或关闭当前折叠；zM，关闭所有折叠；zR，打开所有折叠
- 注意：在.vimrc中增加如下信息即可实现代码折叠：

```
"选择代码折叠类型
set foldmethod=syntax
"启动vim时不要自动折叠代码
set foldlevel=100
```

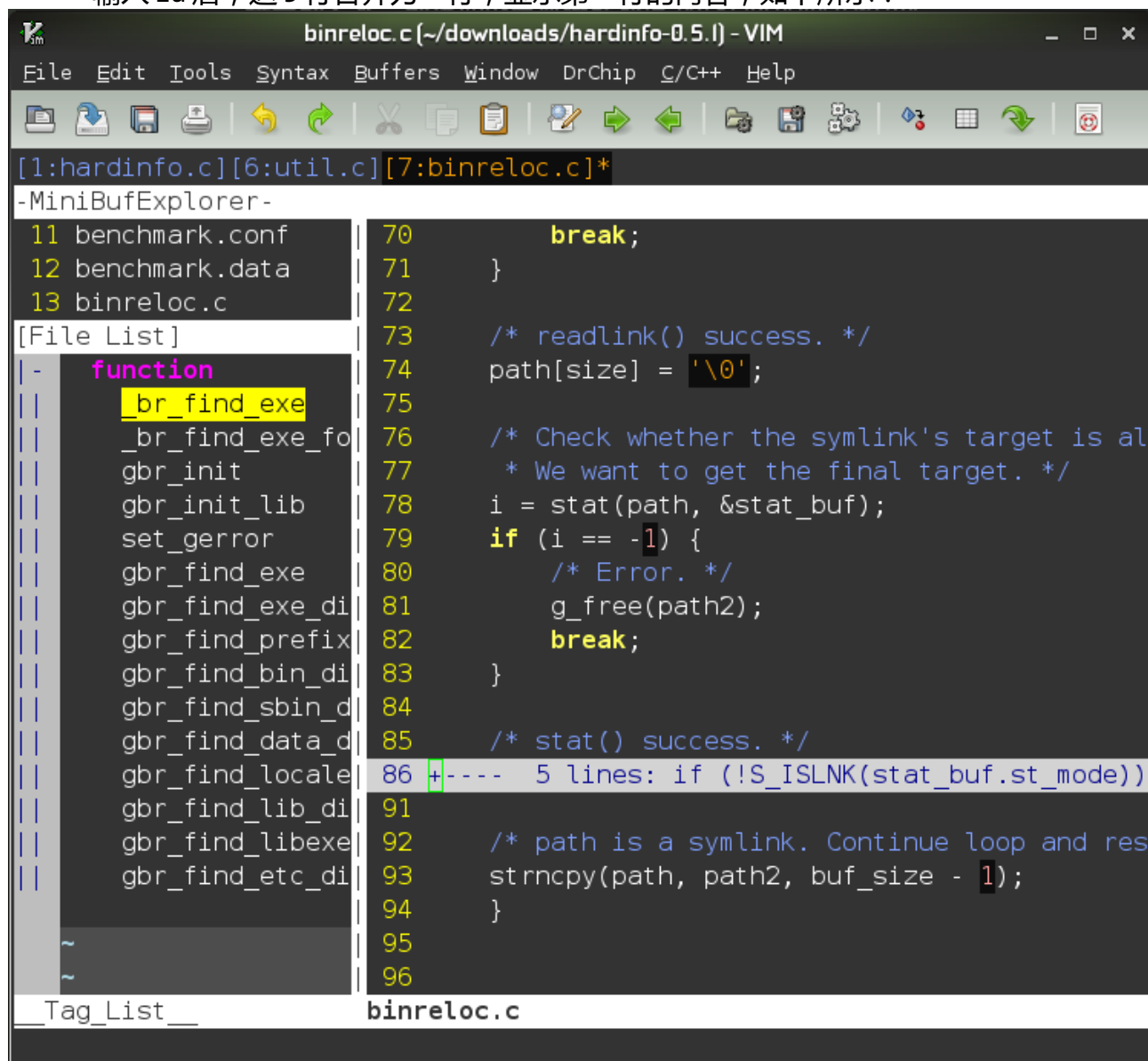
如，当前光标位于87行的if语句块内，该语句块处于正常展开的状态，占据从86到90处共计5行空间，如下：



```
75
76  /* Check whether the symlink's target is al
77   * We want to get the final target. */
78  i = stat(path, &stat_buf);
79  if (i == -1) {
80      /* Error. */
81      g_free(path2);
82      break;
83  }
84
85  /* stat() success. */
86  if (!S_ISLNK(stat_buf.st_mode)) {
87      /* path is not a symlink. Done. */
88      g_free(path2);
89      return path;
90  }
91
92  /* path is a symlink. Continue loop and res
93  strncpy(path, path2, buf_size - 1);
94  }
95
96
97  /* readlink() or stat() failed; this can ha
```

(正常展开的代码)

输入 za 后，这 5 行合并为一行，显示第一行的内容，如下所示：



```
binreloc.c (~/.downloads/hardinfo-0.5.1) - VIM
File Edit Tools Syntax Buffers Window DrChip C/C++ Help

[1:hardinfo.c][6:util.c][7:binreloc.c]*
-MiniBufExplorer-
11 benchmark.conf
12 benchmark.data
13 binreloc.c
[File List]
- function
| _br_find_exe
| _br_find_exe_fo
| gbr_init
| gbr_init_lib
| set_gerror
| gbr_find_exe
| gbr_find_exe_di
| gbr_find_prefix
| gbr_find_bin_di
| gbr_find_sbin_d
| gbr_find_data_d
| gbr_find_locale
| gbr_find_lib_di
| gbr_find_libexe
| gbr_find_etc_di
| ~
| ~
Tag_List binreloc.c

70 break;
71 }
72
73 /* readlink() success. */
74 path[size] = '\0';
75
76 /* Check whether the symlink's target is al
77 * We want to get the final target. */
78 i = stat(path, &stat_buf);
79 if (i == -1) {
80 /* Error. */
81 g_free(path2);
82 break;
83 }
84
85 /* stat() success. */
86 +---- 5 lines: if (!S_ISLNK(stat_buf.st_mode))
91
92 /* path is a symlink. Continue loop and res
93 strncpy(path, path2, buf_size - 1);
94 }
95
96
```

(折叠后的代码)

【工程内查找与替换】

有个名为 iFoo 的全局变量，被工程中 10 个文件引用过，由于你岳母觉得匈牙利命名法严重、异常、绝对以及十分万恶，为讨岳母欢心，不得不将该变量更名为 foo，怎么办？依次打开每个文件，逐一查找后替换？

vim 既然被称为“编辑器之神”，这点请求还是可以满足嘛=。=。vim 自身支持全局替换：先用 vim 内部命令 args 选择要替换的文档，然后用全局命令 argdo 执行替换命令%s。如，要将 vim 的当前目录下所有 cpp 和 h 文件中 iFoo 替换为 foo：

```
:args *.cpp *.h
```

```
:argdo %s/\<iFoo\>/foo/ge | update
```

说明几点：

1) args 是基于当前目录进行文件选取的，可通过 vim 内部命令 pwd、cd 等查看、调整当前目录，注意是 vim 的内部命令，不是 shell 的命令（vim 执行 shell 命令需添加“!”前缀）；

2) 上例中，:args *.cpp *.h 仅选取当前目录而非进入其子目录，若要选取当前目录及其子目录可:args **/*.cpp **/*.h，若选取当前目录及其下一层子目录可:args */*.cpp */*.h；

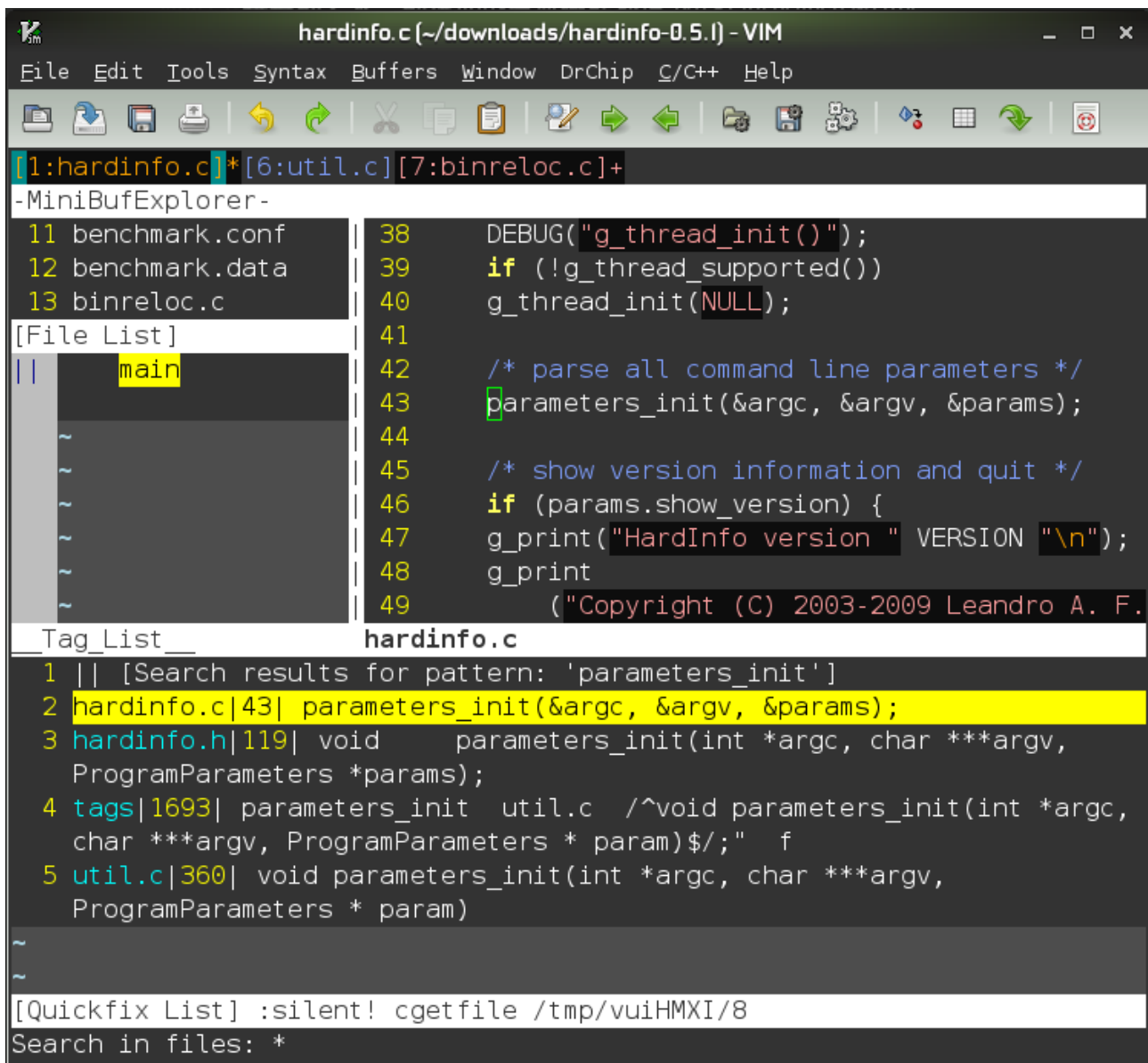
3) 替换命令%s 中的 g 代表对所有文件进行替换操作，e 表示忽略错误，update 用于保存那些有被执行替换操作的文件（不可回退）。

此外，要进行工程内全局查找，可以借助插件实现。

- 插件名：grep
- 操作：在 vim 命令行中输入:Grep 后，grep 插件会依次提示输入待查找关键字、待查找的文件类型，回车即可执行搜索，搜索结果将罗列在 quickfix 中（注，vim 与很多外部命令、插件的交互信息都将在 quickfix 中呈现，这里说到的搜索结果是一个例子，另外一个著名例子为 gcc 的输出信息。可用:cw 命令打开或关闭 quickfix 窗口）。若是要忽然大小写则执行:Grep -i，若是要递归搜索子目录则执行:Grep -r；另外，如果在非空白处执行:Grep 命令，grep 插件自动将当前光标所在单词作为关键字进行搜索。为高效执行搜索操作，可以设定快捷键“;sp”；
- 注意：在.vimrc 中，增加如下配置信息：

```
" 定义快捷键的前缀，即<Leader>
let mapleader=";"
"使用 Grep.vim 插件在工程内全局查找，设置快捷键。快捷键速记法：search in project
nnoremap <Leader>sp :Grep -ir<CR>
```

比如，光标移到 parameters_init 下，输入;sp 后，grep 插件自动提取 parameters_init 为搜索关键字，并给出在哪些类型的文件中内搜索，最后搜索到 4 项匹配结果并显示在 quickfix 中，如下图所示：



```
hardinfo.c (~/.downloads/hardinfo-0.5.1) - VIM
File Edit Tools Syntax Buffers Window DrChip C/C++ Help

[1:hardinfo.c]*[6:util.c][7:binreloc.c]+
-MiniBufExplorer-
11 benchmark.conf
12 benchmark.data
13 binreloc.c
[File List]
|| main
~
~
~
~
~
~
~
~
Tag_List__ hardinfo.c
1 || [Search results for pattern: 'parameters_init']
2 hardinfo.c|43| parameters_init(&argc, &argv, &params);
3 hardinfo.h|119| void parameters_init(int *argc, char ***argv,
ProgramParameters *params);
4 tags|1693| parameters_init util.c /^void parameters_init(int *argc,
char ***argv, ProgramParameters * param)$/;" f
5 util.c|360| void parameters_init(int *argc, char ***argv,
ProgramParameters * param)
~
~
[Quickfix List] :silent! cgetfile /tmp/vuiHMXI/8
Search in files: *
```

(工程内查找)

【一键编译链接及运行】

vim 再强大也只能是个优秀的编辑器而非编译器，它能高效地完成代码编辑工作，但不得不通过其他外部命令实现将代码转换为二进制可执行文件。对于只有单个代码文件的项目来说，无非是保存代码文件、shell 中调用 gcc 编译、链接这样的简单方式即可实现；但，对于动辄几十上百个文件的工程项目，采用这种方式只会把自己逼疯，必须要找到一个实现一键编译的方法。

工程项目一键编译有几个要点，工程自身的管理、编译、查看语法错误，逐一解决即可实现一键编译。简单介绍下，linux 有两种工程代码管理的方式——Makefile 和非 Makefile，采用 Makefile 方式的工程代码管理工具（准确地说，应称之为“构建系

统”) 常见的有 GNU 出品的老牌 autotools、新生代的 CMake (KDE 就是通过 CMake 构建出来的, 非常易用, 洒泪推荐), 采用非 Makefile 方式的构建系统最著名的要数 SCons。我选用 CMake, 由他独立于 vim 生成工程的 Makefile 文件, 解决了第一个工程自身管理的问题; 既然有了 Makefile 文件, 要编译自然是执行 make 命令, 由于 vim 自身支持 make 命令, 直接在 vim 中输入 :make 命令它会调用外部 make 程序读取当前目录中的 Makefile 文件, 完成编译、链接操作, 第二个问题也就解决了; 一次性编译通过的可能性很小, 难免有些语法错误 (语义错误只能靠调试器了), vim 将编译器抛出的错误和警告信息输出到 quickfix 中, 执行 :cw 命令即可显示 quickfix, 第三个问题搞定。编辑、重复以上三步直到编译通过。

说了这么多, 概要之, 先写好整个工程的 Makefile (前面说过, autotools 通过 configure 配置文件生成 Makefile, CMake 通过 CMakeLists.txt 生成 Makefile), 再在 vim 中执行 :make, 最后显示 quickfix。要实现一键编译, 无非是把这几步映射为 vim 的快捷键, 即:

```
nmap <Leader>m :wa<CR>:make<CR>:cw<CR><CR>
```

分解说明下, m 设定快捷键为 m, :wa<CR>保存对所有打开文档的编辑调整, :make<CR>执行 make 命令, :cw<CR>显示 quickfix (仅当有编译错误或警告时), 最后的<CR>消除 make 命令执行完成屏幕上 “Press ENTER or type command to continue” 的输入等待提示信息。

下图是一键编译后的截图, quickfix 窗口显示了编译错误, 光标自动定位到需要你解决的第一个编译错误, 回车后光标自动调整到该错误对应的代码位置。下例中, 编译器提示 24 行 typedef 异常, 立即可以发现, 变量 k 后缺少语句结束符 “;” 。

```

[No Name] -- VIM
15 #include <vector>
16 #include <string>
17 #include <iostream>
18 #include <fstream>
19
20 using namespace std;
21 string k
22
23 // 分配给营业厅的具体IP
24 typedef struct
25 {
26     string          businesshall_name;
27     vector<string>  businesshall_allot_ip;
}
business_hall_ping_ip.cpp 24,1 4%
1 || Scanning dependencies of target business_hall_ping_ip
2 || [100%] Building CXX object CMakeFiles/business_hall_ping_ip.dir/
  business_hall_ping_ip.cpp.o
3 business_hall_ping_ip.cpp|24 col 1| error: expected initializer before 'typedef'
4 business_hall_ping_ip.cpp|28 col 3| error: 'businesshall_allot_ip' does not name
  a type
5 || /root/Desktop/tmp_proj/business_hall_ping_ip.cpp: In function 'int main(int,
  char**)':
6 business_hall_ping_ip.cpp|247 col 9| error: 'businesshall_allot_ip' was not
  declared in this scope
[Quickfix List] :make
```

(一键编译)

另外，你可能会遇到编译完全正常，但在工程目录中无法找到生成的可执行程序，这时很可能代码中存在链接错误。bad news — 如果编译错误，quickfix 窗口会固定在底部，罗列出所有编译过程中的所有错误，如果编译正常（即使存在链接错误），quickfix 窗口会出现“Press ENTER or type command to continue”的输入等待提示信息，前面提过，为了省去手工输入回车，已经在<Leader>m 中绑定了回车符<CR>，换言之，在编译正确但链接错误的情况下，你是无法查看到 quickfix 窗口的；good news — 手工执行外部 make 命令“:!make”，这样，可查看具体是什么链接错误了。

到此，已实现“一键编译”，要实现“一键编译及运行”无非就在刚才的快捷键中追加绑定程序名即可。快捷键改为“;r”，假定生成的可执行程序名为 main：

```
nmap <Leader>r :wa<CR>:make<CR>:cw<CR><CR>:!./main<CR>
```

最后，再次强调实现“一键编译及运行”的几个前提：vim 的当前目录必须为工程目录、事前准备好 Makefile 文件且放于工程目录的根目录、生成的程序必须在工程目录的根目录。

【暂未实现功能】

好了，以上功能基本达到我对 IDE 的预期，满足了？没有，还有几个问题现在还没很好地解决，或者是有思路了，还没来得及实践，记录下来，空了继续探索。如果你清楚，麻烦写信告诉我（yangyang.gnu@gmail.com），多谢多谢 @_@

- 1、集成调试器，实现源码级调试。思路：Pyc1ewn 插件；

2、重启 vim 时恢复到上次编辑环境。包括打开的文件、BUFFER、光标位置等。思路：通过 vim 的会话文件实现。不知道是不是编译 vim 时选项没配好，通过会话实现编辑环境保存都不成功，通过以下变通方式又只能实现将光标定位到关闭前的位置，像打开关闭前的其他文件等需求都无法实现，哎，头痛~~

```
autocmd BufReadPost *
\ if line("'\"") > 0 && line ("'\") <= line("$") |
\   exe "normal g'\") |
\ endif
```

【附一：.vimrc 信息汇总】

```
" 更新时间：2012-12-9

" >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

" >>>>vim 自身相关配置

" 用于语法高亮的配色方案
" colorscheme evening
" colorscheme sean
" colorscheme torte
" colorscheme wombat256
colorscheme molokai

" vim 自身命令行模式智能补全
set wildmenu

" 将外部命令 wmctrl 控制窗口最大化的命令行参数封装成一个 vim 的函数
function Maximize_Window()
    silent !wmctrl -r :ACTIVE: -b add,maximized_vert,maximized_horz
endfunction

" 在右下角显示光标当前位置信息
set ruler
```



```
" 高亮显示当前行
set cursorline

" 禁止光标闪烁
set gcr=a:block-blinkon0

" 设置 gvim 显示字体。
set guifont=YaHei\ Consolas\ Hybrid\ 11.5

" 开启语法高亮功能
syntax enable
" 允许用指定语法高亮配色方案替换默认方案
syntax on

" 设置制表符占用空格数
set tabstop=4
set shiftwidth=4
set noexpandtab

" 开启行号显示
set number

" 开启高亮显示结果
set hlsearch

" 开启实时搜索功能
set incsearch

" 搜索时大小写不敏感
set ignorecase

" 在命令行显示当前输入的命令
set showcmd

" 禁止折行
set nowrap
```

```
" 关闭兼容模式
set nocompatible

" 禁止显示滚动条
set guioptions-=l
set guioptions-=L
set guioptions-=r
set guioptions-=R

" 禁止显示菜单和工具条
set guioptions-=m
set guioptions-=T

" 开启文件类型侦测
filetype on
" 根据侦测到的不同类型加载对应的插件
filetype plugin on
" 根据侦测到的不同类型采用不同的缩进格式
filetype indent on

" 定义快捷键到行首和行尾
nmap lh 0
nmap le $

" 引入操作系统 API、C 库函数、CPP 标准库等汇总 tags
set tags+=/data/misc/software/misc./vim/sys_cpp.tags

" 重新打开文档时光标回到文档关闭前的位置
autocmd BufReadPost *
\ if line("'\"") > 0 && line ("'\") <= line("$") |
\   exe "normal g'\\"" |
\ endif

" 定义快捷键的前缀，即<Leader>
let mapleader=";"

" 定义快捷键关闭当前分割窗口
nmap <Leader>q %
```

[illegible]

- ◆ 以单词为单位移动光标：w、b、W、B
- ◆ 翻页：CTRL - F、CTRL - B
- ◆ 整个文本中移动光标：gg、G、数字G、数字%
- ◆ 当前页中移动光标：H、M、L
- ◆ 移动光标所在行的位置：zz、zt、zb

【搜索】

- ◆ 大小写不敏感：`:set ignorecase`，大小写敏感：`:set noignorecase`
- ◆ 行内搜索：`fx`。X代表要搜索的单个文字（也可以是汉字）。FX为方向搜索。分号重复，逗号反方向重复
- ◆ 整词搜索：`/\<word\>`。整词首尾可拆分搜索
- ◆ 行首、行尾搜索：`/^word`、`/word$`
- ◆ 搜索替代字符：`/ab.de`。“.”代表任意一个字符
- ◆ 替换：`:1,$s/a/b/g`

【自动化命令】

- ◆ 重复上次命令：`.`
- ◆ 撤销上步操作：`u`
- ◆ 重复上步操作：`CTRL - R`。于`.`不同，`CTRL - R`对命令历史记录进行进栈 / 出栈操作

【分割窗口】

- ◆ 新建空白分割窗口：`:new`
- ◆ 在新建空白分割窗口中打开指定文件：`:split filename`
- ◆ 在新建空白分割窗口中显示当前分割窗口内容：`:split`
- ◆ 分割窗口高度调整。增加一行：`CTRL-W, SHIFT-+`；减少一行：`CTRL-W, -`；增加到最大高度：`CTRL-W, SHIFT--`；调整到指定高度：`heightCTRL-W, SHIFT--`

【其他】

- ◆ 取消上次搜索结果高亮显示：`:nohlsearch`
- ◆ 文本另存为：`:saveas file.txt`
- ◆ 多段文本复制：使用寄存器，“ay2j”，“ap”，其中，双引号为寄存器引用前缀，a为自定义寄存器名（只能为一个字母或数字，或代表系统剪贴板的“+”）
- ◆ 宏记录：使用寄存器，qb -> 操作 -> q，其中，q为宏记录开始与结束命令，b为

寄存器，宏回放使用@b。宏回放可加计数器前缀。可“bp打印宏内容，编辑后再“bY。注：复制粘贴和宏记录使用同一套寄存器，所以，同个寄存器的内容即可用于粘贴，也可视为宏记录

- ◆ 选择文本块：v、V、CTRL - V。o、O移动光标在文本块四个角的位置。用I或A命令编辑第一行，再恢复到普通模式下时，被选择块每行首或尾都会有相同新增内容；r命令单个字符替换文本块
- ◆ 操作计数器：数字 - 操作
- ◆ 在线帮助：help keywords
- ◆ 匹配括号：%
- ◆ 恢复选项的默认值：set option&
- ◆ 字母大小写转换：~
- ◆ 转换为html文件：:source \$VIMRUNTIME/syntax/2html.vim, :write main.c.html
- ◆ 在线加载配置文件或插件：:source filepath。如，重新加载配置文件：:source ~/.vimrc
- ◆ 删除光标所在字符到行尾的内容：D
- ◆ 快速向下查找光标所在字符串：*；向上：#
- ◆ 格式化代码：=、>>、<<
- ◆ Vim会在你连续4秒不键入内容时跟磁盘同步一次（内容写入vim临时文件中），或者是连续键入了200个字符之后。这可以通过'updatetime'和'updatecount'两个选项来控制。
- ◆ VIM提供两种方式执行外部命令，一种是“:!cmd”，一种是“!cmd”，前者完全等同于在shell中执行命令，后者相当于同时对命令输入输出重定向，即，将选中的文本块内容作为输入传递给外部命令并用执行结果替换选中文本块。后者用途较为广泛，如，对文本内容排序，可先选中待排序文本块，再键入“!sort”即可，注意，不要键入冒号。也可以仅重定向外部命令输出，即，读取外部命令执行结果：:read !ls，将ls命令执行结果插入当前行。也可以仅重定向外部命令输入，即，将选中文本输入给外部命令执行：:write !wc，将对选中文本块进行计数操作。
- ◆ 查看man信息：先执行:source \$VIMRUNTIME/ftplugin/man.vim。光标移到待查看命令下后键入“\k”后即可在新子窗口中看到man内容，或者“:Man cmd”
- ◆ VIM支持命令行补全，查看全部可键入CTRL-D。如，键入:set i后键入CTRL-D则显示set命令支持的所有以i开头的选项
- ◆ 命令历史窗口：q:，移动光标到指定行回车即可执行该行命令

- 直接打开文件：键入 `gf`，VIM 将当前光标所在字符串视为文件路径并尝试打开编辑该文件。若是绝对路径，则直接打开，若是相对路径，VIM 在 `path` 选项指定的路径范围内进行查找，该 `path` 为 VIM 的一个选项而非 SHELL 的环境变量，默认为 `/usr/include`，可通过 `:set path+=addpath` 或 `:set path-=removepath` 来增删路径。注：若要在分割子窗口中打开可以 `CTRL-W f`
- 重新选中上次选择的文本块：`gv`
- 高效的“操作符”+“文本对象”的组合有六个：
`va()`、`vi()`、`da()`、`di()`、`ca()`、`ci()`。说明下，`v` 是选中，`d` 是删除，`c` 是删除后插入，`a` 是包括结对符在内的整个文本对象，`i` 是结对符内部的文本对象（inner）。如，`va{`选中结对符内的字符串，`di"` 清空结对符内的字符串；
- 安装 vim 中文帮助 <http://vimcdoc.sourceforge.net/>
- 显示当前光标在文档中的位置信息：`CTRL-G`
- 启动 `gvim` 自动最大化。

A. 安装 `wmctrl` 窗口控制命令；

B. 将外部命令 `wmctrl` 控制窗口最大化的命令行参数封装成一个 vim 的函数，并加入到 `.vimrc` 中：

```
function Maximize_Window()  
    silent !wmctrl -r :ACTIVE: -b  
    add,maximized_vert,maximized_horz  
endfunction
```

C. 将 `gvim` 命令名重定义为 `gvi -c "call Maximize_Window()"` 以便使用，并加入 `.bashrc` 中：

```
alias gvim='gvim -c "call Maximize_Window()"'
```