Back to Table of Contents

C/C++ Development Environment for Emacs

In this guide, I will help you to setup an efficient working C/C++ environment. Despite looking long, the setup is short and easy (mostly copy/paste Emacs Lisp code into your `init.el`); most of the guide are explanations and demonstrations of many useful features. Following this guide, you should be able to browse the Linux kernel source tree inside Emacs effortlessly, such as jump to definition/references at cursor, go back and forth between jumping points, finding any file instantly, switching between .h and .c/.cpp.

A few demos:

- Switching between .h and .c/.cpp anywhere in the project like Linux kernel. If more than one file exists, it displays a list of possible candidates. The command collects files of the same names but different across the project:

Table of Contents

```
1  #ifndef __ASM_ALPHA_FPU_H
2  #define __ASM_ALPHA_FPU_H
3
4  #include <asm/special_insns.h>
5  #include <uapi/asm/fpu.h>
6
7  /* The following two functions don't need trapb/excb instructions
8     around the mf_fpcr/mt_fpcr instructions because (a) the kernel
9     never generates arithmetic faults and (b) call_pal instructions
10    are implied trap barriers.  */
11
12 static inline unsigned long
13 rdfpcr(void)
14 {
15     unsigned long tmp, ret;
16
17 #if defined(CONFIG_ALPHA_EV6) || defined(CONFIG_ALPHA_EV67)
18     __asm__ __volatile__ (
19         "ftoit $f0,%0\n\t"
20         "mf_fpcr $f0\n\t"
21         "ftoit $f0,%1\n\t"
22         "itoft %0,$f0"
23         : "=r"(tmp), "=r"(ret));
24 #else
25     __asm__ __volatile__ (
26         "stt $f0,%0\n\t"
27         "mf_fpcr $f0\n\t"
28         "stt $f0,%1\n\t"
29         "ldt $f0,%0"
30         : "=m"(tmp), "=m"(ret));
31 #endif
32
33     return ret;
34 }
35
36 static inline void
37 wrfpcr(unsigned long val)
38 {
39     unsigned long tmp;
40
41 #if defined(CONFIG_ALPHA_EV6) || defined(CONFIG_ALPHA_EV67)
42     __asm__ __volatile__ (
43         "ftoit $f0,%0\n\t"
.../linux/arch/alpha/include/asm/fpu.h  [Ins]                    ( 1, 0) [Top/1.8k] [
Eval: START
```

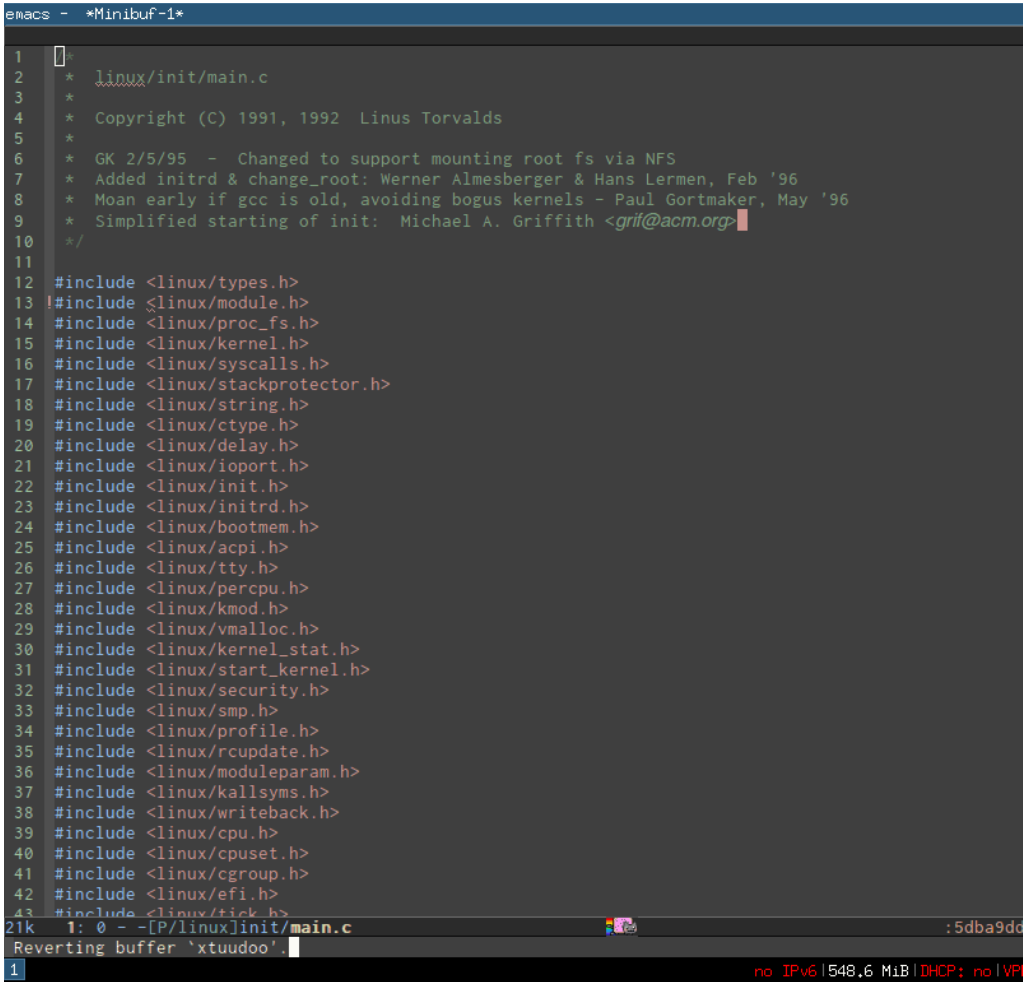
- Jump around Linux kernel source with ease using `helm-gtags`. The demo begins when "START" appears at the bottom:

Table of Contents

```
 * Set up kernel memory allocators
 */
static void __init mm_init(void)
{
    /*
     * page_cgroup requires contiguous pages,
     * bigger than MAX_ORDER unless SPARSEMEM.
     */
    page_cgroup_init_flatmem();
    mem_init();
    kmem_cache_init();
    percpu_init_late();
    pgtable_init();
    vmalloc_init();
}

asmlinkage __visible void __init start_kernel(void)
{
    char * command_line;
    extern const struct kernel_param __start___param[], __stop___param[];

    /*
     * Need to run as early as possible, to initialize the
     * lockdep hash:
     */
    lockdep_init();
    smp_setup_processor_id();
    debug_objects_early_init();

    /*
     * Set up the the initial canary ASAP:
     */
    boot_init_stack_canary();

    cgroup_init_early();

    local_irq_disable();
    early_boot_irqs_disabled = true;

/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
    boot_cpu_init();
    page_address_init();
    pr_notice("%s", linux_banner);
    setup_arch(&command_line);
    mm_init_owner(&init_mm, &init_task);
    mm_init_cpumask(&init_mm);
    setup_command_line(command_line);
    setup_nr_cpu_ids();
    setup_per_cpu_areas();
    smp_prepare_boot_cpu(); /* arch-specific boot-cpu hooks */
~/linux/init/main.c  [Ins]                    (490, 7) [52%/23k] [C/l] company [Git:m
Eval: START
```

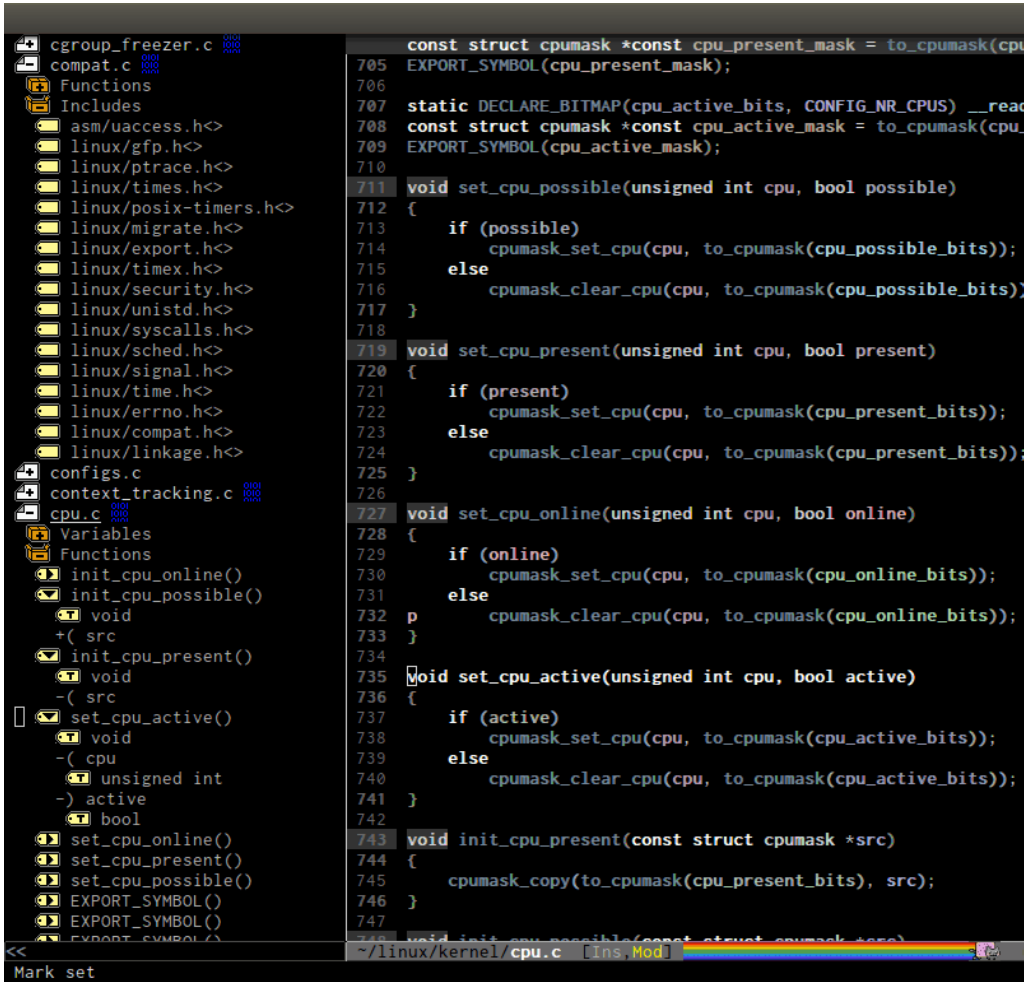
Table of Contents

- Interactive outline tree using `moo-jump-local` from function-args package:



- Static outline tree as a file browser:

Table of Contents



- Symbol references:

Table of Contents

```
            .name = "next-server",
1852        .description = "TFTP server",
1853        .tag = DHCP_EB_SIADDR,
1854        .type = &setting_type_ipv4,
1855    };
1856
1857    /** Filename setting */
1858    struct setting filename_setting __setting ( SETTING_BOOT
        $) = {
1859        .name = "filename",
1860        .description = "Boot filename",
1861        .tag = DHCP_BOOTFILE_NAME,
1862        .type = &setting_type_string,
1863    };
1864
1865    /** Root path setting */
1866    struct setting root_path_setting __setting ( SETTING_SANB
        $OOT ) = {
1867        .name = "root-path",
1868        .description = "SAN root path",
1869        .tag = DHCP_ROOT_PATH,
1870        .type = &setting_type_string,
1871    };
1872
1873    /** Username setting */
1874    struct setting username_setting __setting ( SETTING_AUTH
        $) = {
1875        .name = "username",
1876        .description = "User name",
1877        .tag = DHCP_EB_USERNAME,
1878        .type = &setting_type_string,
1879    };
1880
1881    /** Password setting */
1882    struct setting password_setting __setting ( SETTING_AUTH
        $) = {
1883        .name = "password",
1884        .description = "Password",
1885        .tag = DHCP_EB_PASSWORD,
1886        .type = &setting_type_string,
1887    };
1888
1889    /** Priority setting */
1890    struct setting priority_setting __setting ( SETTING_MISC
.../enux/src/ipxe/ipxe/src/core/settings.c  [Ins] (First workgro
Beginning of buffer
```

```
/home/xtuudoo/enux/src/ipxe/ipxe/src/arch/i386/interface/vmware/guestin
  [+] static int guestinfo_fetch_type (struct settings settings,struct
$ype type,void data,class size_t len,int found)
  [+] static int guestinfo_fetch (struct settings settings,struct setti
$t len)
/home/xtuudoo/enux/src/ipxe/ipxe/src/core/nvo.c
  [+] static int nvo_applies (struct settings settings,class __unused,s
  [+] struct settings settings
  [-] static int nvo_store (struct settings settings,struct setting set
$t len)
    if ( ( rc = dhcpopt_store ( &nvo->dhcpopts, setting->tag,
  [-] struct settings settings
    static int nvo_fetch ( struct settings *settings, struct setting *s
  [-] static int nvo_fetch (struct settings settings,struct setting set
    return dhcpopt_fetch ( &nvo->dhcpopts, setting->tag, data, len );
/home/xtuudoo/enux/src/ipxe/ipxe/src/core/settings.c
  [+] struct generic_setting {}
  [+] static struct generic_setting find_generic_setting (struct generi
$ng setting)
  [+] int generic_settings_store (struct settings settings,struct setti
$ size_t len)
  [+] int generic_settings_fetch (struct settings settings,struct setti
$t len)
  [+] int setting_applies (struct settings settings,struct setting sett
  [+] int store_setting (struct settings settings,struct setting settin
$en)
  [+] static int fetch_setting_and_origin (struct settings settings,str
$ngs origin,void data,class size_t len)
  [+] int fetch_setting (struct settings settings,struct setting settin
  [+] struct settings fetch_setting_origin (struct settings settings,st
  [+] int fetch_setting_len (struct settings settings,struct setting se
  [+] int fetch_setting_copy (struct settings settings,struct setting s
  [+] int fetch_string_setting (struct settings settings,struct setting
$len)
  [+] int fetch_string_setting_copy (struct settings settings,struct se
  [+] int fetch_ipv4_array_setting (struct settings settings,struct set
$unsigned int count)
  [+] int fetch_ipv4_setting (struct settings settings,struct setting s
  [+] int fetch_int_setting (struct settings settings,struct setting se
  [+] int fetch_uint_setting (struct settings settings,struct setting s
  [+] long fetch_intz_setting (struct settings settings,struct setting
  [+] unsigned long fetch_uintz_setting (struct settings settings,struc
  [+] int fetch_uuid_setting (struct settings settings,struct setting s
  [+] int setting_cmp (struct setting a,struct setting b)
.../enux/src/ipxe/ipxe/src/core/*Symref setting  [Ins,Mod,RO] (First wo
```

- Code completion 1:

Table of Contents

```
1 #include <boost/asio.hpp>
2
3 int main(int argc, char *argv[])
4 {
5     
6     return 0;
7 }
~/test.cpp  [Ins,Mod] (test)          ( 5, 4) [All/83] [C++/l] company
Eval: START
```

Table of Contents

- Code completion 2:

```
1 #include <algorithm>
2 #include <string>
3 #include <iostream>
4 #include <map>
5 #include "coloring_solver.h"
6 #include <stdio.h>
7
8 Answer ColoringSolver::solve() {
9     using namespace std;
10
11     p
12
13     set<int> used_colors, viable_colors;
14     map<int, set<int> > original;
15
16     for (auto& u:vertices)
17     {
18         for (auto& v:u.end_vertices)
19         {
20             original[u.id].insert(v.second->id);
21         }
22     }
23
24     // for (auto& o : original)
25     // {
26     //     cout << "origin i: " << o.first << endl;
27     //     cout << "origin end vertices: ";
28     //     for (auto& id : o.second )
29     //     {
30     //         cout << id << ' ';
31     //     }
32     //     cout << endl;
33     // }
34
35     sort(vertices.begin(), vertices.end(), [] (const Vertex& a, const Vertex& b) {
36             return a.end_vertices.size() > b.end_vertices.size();
37         });
38
39     for (auto& u : vertices )
40     {
41         for (auto& v_idx:original[u.id])
42         {
43             auto it = find_if (vertices.begin(), vertices.end(), [&v_idx] (const Vertex& o) -> bool {
44                     return o.id == v_idx;
.../discrete_optimization/hw2/coloring/coloring_solver.cpp GG [Ins]  (11, 5) [Top/9.4k] [C++/l] [Git:mas
call to non-static member function without an object argument
```

- Header completion:

Table of Contents

- Show function interface and variable definition at the bottom:

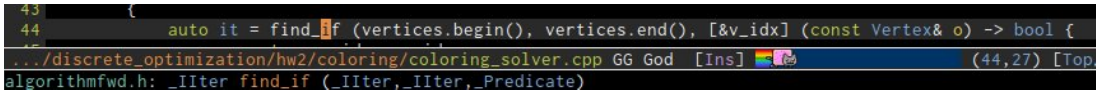
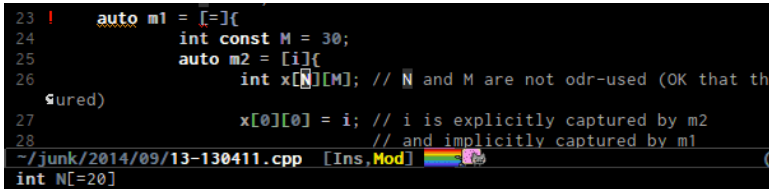


Table of Contents



- Show current function your cursor is inside at the top:

Table of Contents

```
     * Set up kernel memory allocators
464   */
465  static void __init mm_init(void)
466  {
467      /*
468       * page_cgroup requires contiguous pages,
469       * bigger than MAX_ORDER unless SPARSEMEM.
470       */
471      page_cgroup_init_flatmem();
472      mem_init();
473      kmem_cache_init();
474      percpu_init_late();
475      pgtable_init();
476      vmalloc_init();
477  }
478
479  asmlinkage __visible void __init start_kernel(void)
480  {
481      char * command_line;
482      extern const struct kernel_param __start___param[], __stop___param[];
483
484      /*
485       * Need to run as early as possible, to initialize the
486       * lockdep hash:
487       */
488      lockdep_init();
489      smp_setup_processor_id();
490      debug_objects_early_init();

492      /*
493       * Set up the the initial canary ASAP:
494       */
495      boot_init_stack_canary();
496
497      cgroup_init_early();
498
499      local_irq_disable();
500      early_boot_irqs_disabled = true;
501
502  /*
503   * Interrupts are still disabled. Do necessary setups, then
504   * enable them
505   */
506      boot_cpu_init();
507      page_address_init();
508      pr_notice("%s", linux_banner);
509      setup_arch(&command_line);
510      mm_init_owner(&init_mm, &init_task);
511      mm_init_cpumask(&init_mm);
512      setup_command_line(command_line);
513      setup_nr_cpu_ids();
514      setup_per_cpu_areas();
515      smp_prepare_boot_cpu(); /* arch-specific boot-cpu hooks */
~/linux/init/main.c  [Ins] (test)                    (479, 0) [52%/23k]
Eval: START
```

- Compilation support:

Table of Contents

Table of Contents

```
/home/tuhdo/Dropbox/Projects/s3183594_s3245863_Le Duc Tuan_Do Hoang Tu_COSC2131_A2/Terminal:
.
..
.svn
basketball.txt
chemical_substance.txt
#Crossword.cpp#
Crossword.cpp
Crossword.h
CrosswordLib.h
Crossword.o
Letter.cpp
Letter.h
Letter.o
main.cpp
main.o
Makefile
Position.cpp
Position.h
Position.o
sample_data1.txt
sample_data2.txt
sample_data3.txt
sample_data4.txt
science.txt
solarsystem_crossword_puzzle.txt
#Word.cpp#
Word.cpp
Word.h
Word.o
.../s3183594_s3245863_Le Duc Tuan_Do Hoang Tu_COSC2131_A2/Terminal/Terminal  [Ins,Mod,RO] (test)  ( 5, 2
Eval: START
```

Table of Contents

- Beautiful compile output:

```
checking for cp... cp
checking for ln... ln
checking for tar... tar
checking for rpmbuild... rpmbuild
checking for sed... sed
checking for find... find
checking for xargs... xargs
checking for dirname... dirname
checking for grep that handles long lines and -e... /usr/bin/grep
checking for egrep... /usr/bin/grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking zlib.h usability... yes
checking zlib.h presence... yes
checking for zlib.h... yes
checking for inflateInit_ in -lz... yes
checking lzma.h usability... no
checking lzma.h presence... no
checking for lzma.h... no
checking xenctrl.h usability... no
checking xenctrl.h presence... no
checking for xenctrl.h... no
configure: creating ./config.status
config.status: creating Makefile
config.status: creating include/config.h
+ make
gcc -fmessage-length=0 -O2 -Wall -D_FORTIFY_SOURCE=2 -fstack-protector -funwind-tables -fasynchronou
nclude -I./util_lib/include -Iinclude/ -I./kexec/arch/x86_64/libfdt -I./kexec/arch/x86_64/include  -
kexec/kexec.c: In function 'main':
kexec/kexec.c:1229: warning: implicit declaration of function 'xen_balloon_up'
 /local/xtuudoo/enux/*compilation*  [Ins,Mod,RO]  (1473, 0) [26%/304k] [Compilation] company
```

- Fancy GDB debugging:

Table of Contents

```
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bufbomb...done.
(gdb) b test
Breakpoint 1 at 0x400ed0: file bufbomb.c, line 103.
(gdb) r -u 2866
Starting program: /home/tuhdo/course-materials/lab3/bufbomb -u 2866
(gdb)
```

```
Locals Registers
unsigned long long          val             <optimized out>
volatile unsigned long long local           140737351856328
char *                      variable_length <optimized out>
```

```
~/course-materials/lab3/*gud-bufbomb*  [Ins,Mod] (test)
     void fizz(int arg1, char arg2, long arg3, char* arg4, short arg5, short arg6, unsign
 79          printf("Misfire: You called fizz(0x%llx)\n", val);
 80      }
 81      exit(0);
 82  }
 83  /* $end fizz-c */

 84
 85  /* $begin bang-c */
 86  unsigned long long global_value = 0;
 87
 88  void bang(unsigned long long val)
 89  {
 90      entry_check(2);  /* Make sure entered this function properly */
 91      if (global_value == cookie) {
 92          printf("Bang!: You set global_value to 0x%llx\n", global_value);
 93          validate(2);
 94      } else {
 95          printf("Misfire: global_value = 0x%llx\n", global_value);
 96      }
 97      exit(0);
 98  }
 99  /* $end bang-c */
100
101  /* $begin boom-c */
102  void test()
103 {
```

```
~/course-materials/lab3/*locals of bufbomb*  [Ins,Mod,RO] (
Username: 2866
96
95
64
7e
ac
a
ce
8c
91
5f
1d
55
48
4c
Cookie: 0x4c48551d5f918cce
```

```
~/course-materials/lab3/bufbomb.c  [Ins] (test)                    (103, 0)
0 in test of bufbomb.c:103
1 in launch of bufbomb.c:343
2 in main of bufbomb.c:454
```

```
~/course-materials/lab3/*input/output of bufbomb*  [Ins,Mod
Breakpoints Threads
Num Type       Disp Enb Addr               Hits What
1   breakpoint keep y   0x0000000000400ed0 1    in test of
```

```
~/course-materials/lab3/*stack frames of bufbomb*  [Ins,Mod,RO] (test)
Eval: START
```

```
~/course-materials/lab3/*breakpoints of bufbomb*  [Ins,Mod,
```

Table of Contents

- Getting man pages for symbol at cursor:

```
DEFUN ("dump-emacs", Fdump_emacs, Sdump_emacs, 2, 2, 0,
2143      Meanwhile, my_edata is not valid on Windows.  */
2144    {
2145      extern char my_edata[];
2146      memory_warnings (my_edata, malloc_warning);
2147    }
2148  #endif /* not WINDOWSNT */
2149  #endif /* not SYSTEM_MALLOC */
2150  #ifdef DOUG_LEA_MALLOC
2151    malloc_state_ptr = malloc_get_state ();
2152  #endif
2153
2154    unexec (SSDATA (filename), !NILP (symfile) ? SSDATA (symfile) : 0);
2155
2156  #ifdef DOUG_LEA_MALLOC
2157    free (malloc_state_ptr);
2158  #endif
2159
2160  #ifdef WINDOWSNT
2161    Vlibrary_cache = Qnil;
2162  #endif
2163  #ifdef HAVE_WINDOW_SYSTEM
2164    reset_image_types ();
2165  #endif
2166
2167    Vpurify_flag = tem;
2168
2169    return unbind_to (count, Qnil);
2170  }
2171
2172  #endif /* not CANNOT_DUMP */
2173  ^L
2174  #if HAVE_SETLOCALE
2175  /* Recover from setlocale (LC_ALL, "").  */
2176  void
2177  fixup_locale (void)
2178  {
2179    /* The Emacs Lisp reader needs LC_NUMERIC to be "C",
2180       so that numbers are read and printed properly for Emacs Lisp.  */
2181    setlocale (LC_NUMERIC, "C");
2182  }
2183
2184  /* Set system locale CATEGORY, with previous locale *PLOCALE, to
2185     DESIRED_LOCALE.  */
~/Downloads/emacs/src/emacs.c  [Ins]        (2157, 5) [82%/77k] [C/l] company [Git-master]  [dtrt-indent adjus
Eval: START
```

Please remember that there's a table of content on the top right corner and you can use it to navigate this guide. If you feel your internet connection is too slow for my guide, you can always clone and read it offline: https://github.com/tuhdo/tuhdo.github.io

I added an Emacs repository that is properly configured for demonstration purpose. You can clone and play with it and you can ignore all the Elisp code for setting up throughout the guide:

Table of Contents

```
git clone https://github.com/tuhdo/emacs-c-ide-demo.git ~/.emacs.d
```

Remember to backup your `~/.emacs.d` elsewhere. In the demo repository, I already installed both `ggtags` and `helm-gtags`. `helm-gtags` is enabled by default. If you want to use `ggtags`, comment this line in `init.el`:

```
(require 'setup-helm-gtags)
```

If you don't like to use Helm, also uncomment `(require 'setup-helm)` and restart Emacs, because I made some global key bindings to some commands that even if `helm-mode` is disabled, such commands still uses Helm interface. If you want to learn Helm, follow my guide.

And uncomment this line:

```
;; (require 'setup-ggtags)
```

Source code navigation

Prerequisite:

- Know how to use `package.el` and MELPA. If you don't know how to use, read the guide How to use Emacs package manager.
- Have GNU Global installed. Download the source here, or you can install it from the package manager of your OS (Linux distribution or Mac OS). For Windows users, download the Win32 port.

  Warning: if you install from a package manager, check that you don't get an outdated version (helm-gtags might not fully work otherwise).

- Install ggtags. After installing `ggtags` from MELPA, add this code snippet to setup `ggtags` and key bindings:

```
(require 'ggtags)
(add-hook 'c-mode-common-hook
          (lambda ()
            (when (derived-mode-p 'c-mode 'c++-mode 'java-mode 'asm-mode)
              (ggtags-mode 1))))

(define-key ggtags-mode-map (kbd "C-c g s") 'ggtags-find-other-symbol)
(define-key ggtags-mode-map (kbd "C-c g h") 'ggtags-view-tag-history)
(define-key ggtags-mode-map (kbd "C-c g r") 'ggtags-find-reference)
(define-key ggtags-mode-map (kbd "C-c g f") 'ggtags-find-file)
(define-key ggtags-mode-map (kbd "C-c g c") 'ggtags-create-tags)
```

Table of Contents

```
(define-key ggtags-mode-map (kbd "C-c g u") 'ggtags-update-tags)

(define-key ggtags-mode-map (kbd "M-,") 'pop-tag-mark)
```

- Or, helm + helm-gtags. Helm is awesome and if you are going to use Helm, please read the Helm guide. Remember to setup `Helm` before using `helm-gtags`. You can use this sample configuration. When includes the above file in your `~/.emacs.d`, remember to add `(require 'setup-helm)` to your `init.el`.

```
(setq
 helm-gtags-ignore-case t
 helm-gtags-auto-update t
 helm-gtags-use-input-at-cursor t
 helm-gtags-pulse-at-cursor t
 helm-gtags-prefix-key "\C-cg"
 helm-gtags-suggested-key-mapping t
 )

(require 'helm-gtags)
;; Enable helm-gtags-mode
(add-hook 'dired-mode-hook 'helm-gtags-mode)
(add-hook 'eshell-mode-hook 'helm-gtags-mode)
(add-hook 'c-mode-hook 'helm-gtags-mode)
(add-hook 'c++-mode-hook 'helm-gtags-mode)
(add-hook 'asm-mode-hook 'helm-gtags-mode)

(define-key helm-gtags-mode-map (kbd "C-c g a") 'helm-gtags-tags-in-this-function)
(define-key helm-gtags-mode-map (kbd "C-j") 'helm-gtags-select)
(define-key helm-gtags-mode-map (kbd "M-.") 'helm-gtags-dwim)
(define-key helm-gtags-mode-map (kbd "M-,") 'helm-gtags-pop-stack)
(define-key helm-gtags-mode-map (kbd "C-c <") 'helm-gtags-previous-history)
(define-key helm-gtags-mode-map (kbd "C-c >") 'helm-gtags-next-history)
```

Before using the `ggtags` or `helm-gtags`, remember to create a GTAGS database by running `gtags` at your project root in terminal:

```
$ cd /path/to/project/root
$ gtags
```

After this, a few files are created:

Table of Contents

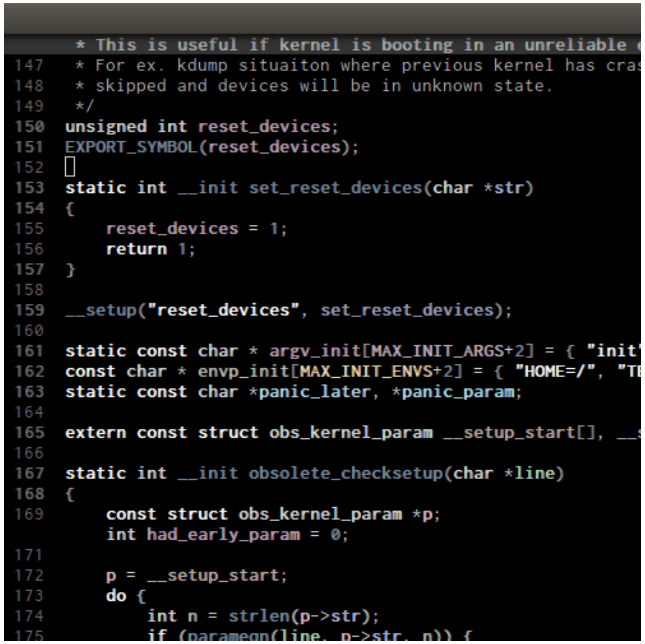
```
$ ls G*
GPATH   GRTAGS  GTAGS
```

- GTAGS: definition database
- GRTAGS: reference database
- GPATH: path name database

If you use `ggtags`, you have a command for creating GTAGS database, that is `ggtags-create-tags`; this is recommended way when using `ggtags`, to let it know where the project root is.

Basic movements

- C-M-f runs `forward-sexp`, move forward over a balanced expression that can be a pair or a symbol. Demo:



- C-M-b runs `backward-sexp`, move backward over a balanced expression that can be a pair or a symbol. Demo:

Table of Contents

```
 * This is useful if kernel is booting in an unreliable e
 * For ex. kdump situaiton where previous kernel has cras
 * skipped and devices will be in unknown state.
 */
unsigned int reset_devices;
EXPORT_SYMBOL(reset_devices);

static int __init set_reset_devices(char *str)
{
    reset_devices = 1;
    return 1;
}

__setup("reset_devices", set_reset_devices);

static const char * argv_init[MAX_INIT_ARGS+2] = { "init"
const char * envp_init[MAX_INIT_ENVS+2] = { "HOME=/", "TE
static const char *panic_later, *panic_param;

extern const struct obs_kernel_param __setup_start[], __s

static int __init obsolete_checksetup(char *line)
{
    const struct obs_kernel_param *p;
    int had_early_param = 0;

    p = __setup_start;
    do {
        int n = strlen(p->str);
        if (parameqn(line, p->str, n)) {
```

- C-M-k runs `kill-sexp`, kill balanced expression forward that can be a pair or a symbol. Demo:

Table of Contents

```
 * This is useful if kernel is booting in an unreliable e
 * For ex. kdump situaiton where previous kernel has cras
 * skipped and devices will be in unknown state.
 */
unsigned int reset_devices;
EXPORT_SYMBOL(reset_devices);

static int __init set_reset_devices(char *str)
{
    reset_devices = 1;
    return 1;
}

__setup("reset_devices", set_reset_devices);

static const char * argv_init[MAX_INIT_ARGS+2] = { "init"
const char * envp_init[MAX_INIT_ENVS+2] = { "HOME=/", "TE
static const char *panic_later, *panic_param;

extern const struct obs_kernel_param __setup_start[], __s

static int __init obsolete_checksetup(char *line)
{
    const struct obs_kernel_param *p;
    int had_early_param = 0;

    p = __setup_start;
    do {
        int n = strlen(p->str);
        if (parameqn(line, p->str, n)) {
```

- C-M-<SPC> or C-M-@ runs `mark-sexp`, put mark after following expression that can be a pair or a symbol. Demo:

Table of Contents

```
    static int __init obsolete_checksetup(char *line)
168 {
169     const struct obs_kernel_param *p;
170     int had_early_param = 0;
171
172     p = __setup_start;
173     do {
174         int n = strlen(p->str);
175         if (parameqn(line, p->str, n)) {
176             if (p->early) {
177                 /* Already done in parse_early_param?
178                  * (Needs exact match on param part).
179                  * Keep iterating, as we can have early
180                  * params and __setups of same names 8( */
181                 if (line[n] == '\0' || line[n] == '=')
182                     had_early_param = 1;
183             } else if (!p->setup_func) {
184                 pr_warn("Parameter %s is obsolete, ignored\n",
185                     p->str);
186                 return 1;
187             } else if (p->setup_func(line + n))
188                 return 1;
189         }
190         p++;
        }while (p < __setup_end);
192
193     return had_early_param;
194 }
195
196 /*
197  * This should be approx 2 Bo*oMips to start (note initial shift), an
198  * still work even if initially too large, it will just take slightly
199  */
200 unsigned long loops_per_jiffy = (1<<12);
201
202 EXPORT_SYMBOL(loops_per_jiffy);
```

- C-M-a runs `beginning-of-defun`, which moves point to beginning of a function. Demo:

Table of Contents

```
static int __init obsolete_checksetup(char *line)
    p = __setup_start;
    do {
        int n = strlen(p->str);
        if (parameqn(line, p->str, n)) {
            if (p->early) {
                /* Already done in parse_early_param?
                 * (Needs exact match on param part).
                 * Keep iterating, as we can have early
                 * params and __setups of same names 8( */
                if (line[n] == '\0' || line[n] == '=')
                    had_early_param = 1;
            } else if (!p->setup_func) {
                pr_warn("Parameter %s is obsolete, ignored\n",
                    p->str);
                return 1;
            } else if (p->setup_func(line + n))
                return 1;
        }
        p++;
    } while (p < __setup_end);

    return had_early_param;
}

/*
 * This should be approx 2 Bo*oMips to start (note initial shift), an
 * still work even if initially too large, it will just take slightly
 */
unsigned long loops_per_jiffy = (1<<12);

EXPORT_SYMBOL(loops_per_jiffy);

static int __init debug_kernel(char *str)
{
    console_loglevel = 10;
    return 0;
}

static int __init quiet_kernel(char *str)
{
    console_loglevel = 4;
    return 0;
}
~/linux/init/main.c  [Ins] (First workgroup)
Eval: START
```

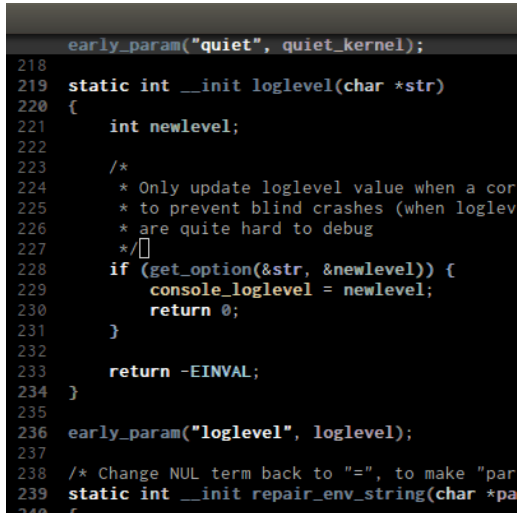
- C-M-e runs `end-of-defun`, which moves point to end of a function. Demo:

Table of Contents

```
167 static int __init obsolete_checksetup(char *line)
168 {
169     const struct obs_kernel_param *p;
170     int had_early_param = 0;
171
172     p = __setup_start;
173     do {
174         int n = strlen(p->str);
175         if (parameqn(line, p->str, n)) {
176             if (p->early) {
177                 /* Already done in parse_early_param?
178                  * (Needs exact match on param part).
179                  * Keep iterating, as we can have early
180                  * params and __setups of same names 8( */
181                 if (line[n] == '\0' || line[n] == '=')
182                     had_early_param = 1;
183             } else if (!p->setup_func) {
184                 pr_warn("Parameter %s is obsolete, ignored\n",
185                     p->str);
186                 return 1;
187             } else if (p->setup_func(line + n))
188                 return 1;
189         }
190         p++;
191     } while (p < __setup_end);
192
193     return had_early_param;
194 }
195
196 /*
197  * This should be approx 2 Bo*oMips to start (note initial shift), an
198  * still work even if initially too large, it will just take slightly
199  */
200 unsigned long loops_per_jiffy = (1<<12);
201
202 EXPORT_SYMBOL(loops_per_jiffy);
203
204 static int __init debug_kernel(char *str)
205 {
206     console_loglevel = 10;
207     return 0;
208 }
209
~/linux/init/main.c  [Ins] (First workgroup)
Eval: START
```

- C-M-h runs `mark-defun`, which put a region around whole current or following function. Demo:

Table of Contents



Basic concepts of tag

A tag is a name of an entity in source code. An entity can be a variable, a method definition, an include-operator... A tag contains several information such as name of the tag (the name of the variable, class, method), location of this tag in source code and which file it belongs to. As an example, GNU Global generates three tag databases:

- GTAGS: definition database
- GRTAGS: reference database
- GPATH: path name database

A definition of a tag is where a tag is implemented. For example, a function definition is the body where it is actually implemented, or a variable definition is where the type and its property (i.e static) is specified.

A reference of a tag is where a tag is used in a source tree, but not where it is defined.

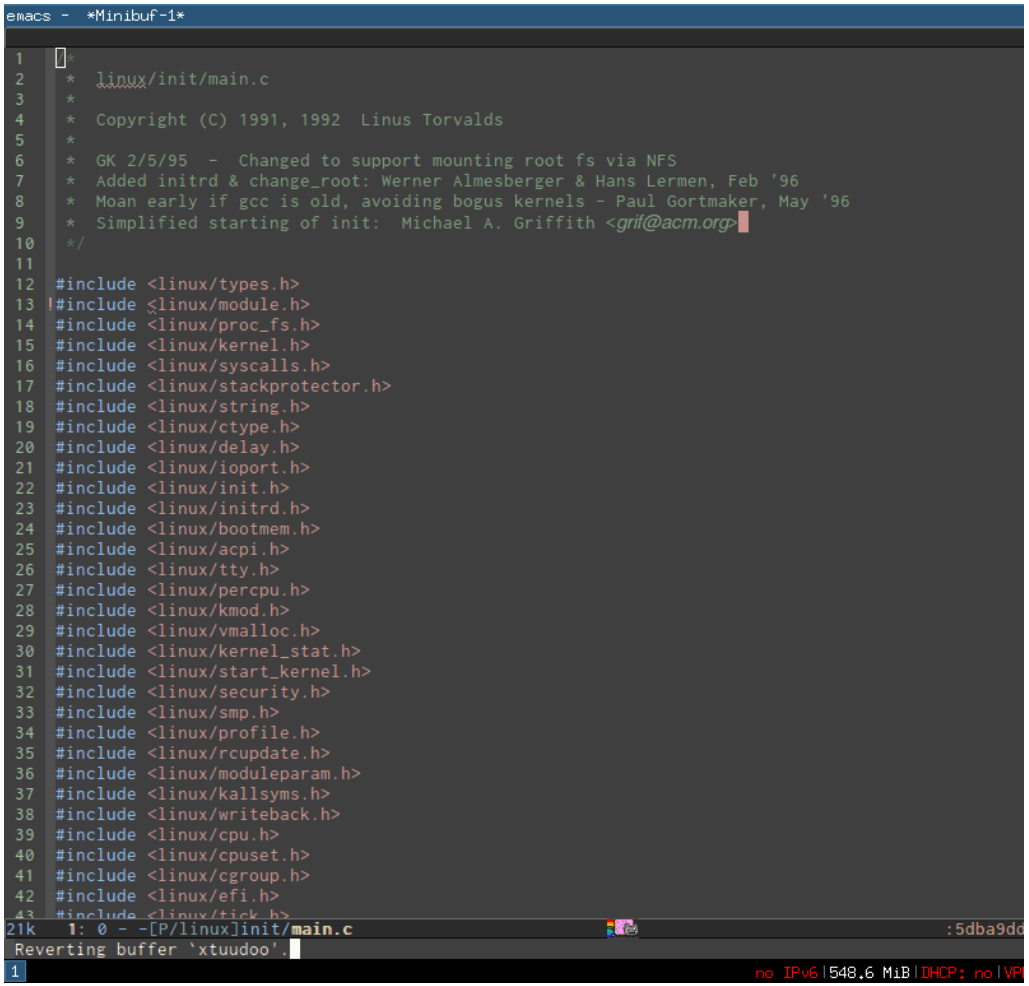
Find definitions in current buffer

The Imenu facility offers a way to find the major definitions, such as function definitions, variable definitions in a file by name. `ggtags` can integrate Imenu:

```
(setq-local imenu-create-index-function #'ggtags-build-imenu-index)
```

Table of Contents

If you use Helm, use `moo-jump-local` from function-args package. You can use it as an outline tree like in other IDEs. Here is a demo:



Find definitions in project

- Using `gtags`: by default, `M-.` runs `ggtags-find-tag-dwim` when `ggtags-mode` is enabled. The command `ggtags-find-tag-dwim` jump to tag base on context:

Table of Contents

- If the tag at point is a definition, `ggtags` jumps to a reference. If there is more than one reference, it displays a list of references.
- If the tag at point is a reference, `ggtags` jumps to tag definition.
- If the tag at point is an include header, it jumps to that header.

You can jump back to original location where you invoked `ggtags-find-tag-dwim` by `M-,`, which runs `pop-tag-mark` (if you follow my key bindings).

You can also find arbitrary tag definition when invoking `M-.` on blank space. A prompt asks you for tag pattern, which is a regexp.

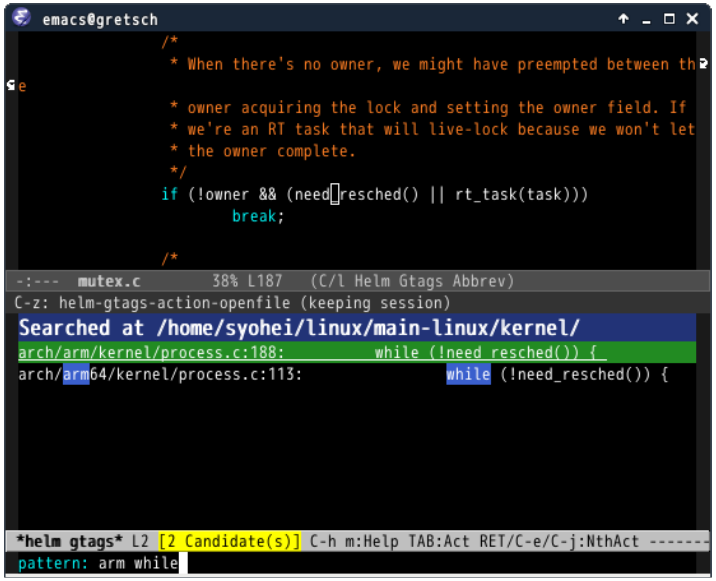
If `ggtags` gives you a list of candidates, you can use `M-n` to move to next candidate and `M-p` to move back previous candidate. Use `M-g s` to invoke Isearch on candidate buffer list.

Table of Contents

(screenshot taken from ggtags)

- Using `helm-gtags`: If key bindings are properly setup as above, `M-.` runs `helm-gtags-dwim`, which behaves the same as `ggtags-find-tag-dwim`. Similarly, you jump back to original location by using `M-,`, which runs `tags-loop-continue` (Emacs default).

Table of Contents



(screenshot taken from helm-gtags)

You can also find arbitrary tag definition when invoking `M-.` on blank space. A prompt asks you for tag pattern, which is a regexp.

`helm-gtags` provides a really nice feature that uses Helm to display all available tags in a project and incrementally filtering, and is really fast using `helm-gtags-select`, which is bound to `C-j` in my setup above. This is useful when you want to explore tags in unfamiliar project. Demo:

Table of Contents

```
early_param("quiet", quiet_kernel);

static int __init loglevel(char *str)
{
    int newlevel;

    /*
     * Only update loglevel value when a correct setting was passed,
     * to prevent blind crashes (when loglevel being set to 0) that
     * are quite hard to debug
     */
    if (get_option(&str, &newlevel)) {
        console_loglevel = newlevel;
        return 0;
    }

    return -EINVAL;
}

early_param("loglevel", loglevel);

/* Change NUL term back to "=", to make "param" the whole string. */
static int __init repair_env_string(char *param, char *val, const char *unused)
{
    if (val) {
        /* param=val or param="val"? */
        if (val == param+strlen(param)+1)
            val[-1] = '=';
        else if (val == param+strlen(param)+2) {
            val[-2] = '=';
            memmove(val-1, val, strlen(val)+1);
            val--;
        } else
            BUG();
    }
    return 0;
}

/*
 * Unknown boot options get handed to init, unless they look like
 * unused parameters (modprobe will find them in /proc/cmdline).
 */
static int __init unknown_bootoption(char *param, char *val, const char *unused)
{
~/linux/init/main.c  [Ins,Mod] (First workgroup)                    (235
Eval: START
```

Find references in project

- Using `ggtags`: Either run `ggtags-find-tag-dwim` or `ggtags-find-reference`, which only finds references.
- Using `helm-gtags`: Either run `helm-gtags-dwim` or `helm-gtags-find-rtags`, bound to C-c g r, which only finds references. Note that for `helm-gtags-find-rtags`:

Table of Contents

- if point is inside a function, the prompt will be default to the function name.
- If point is on a function, it lists references of that functions immediately.
- If point is on a variable, `helm-gtags-find-rtags` won't have any effect. You should use `helm-gtags-find-symbol`, which is bound to C-c g s.

Find functions that calls function is actually a special case of finding references. That is, you gather references for a function.

Find functions that current functions call

If you want to list all the functions that the current function - the function that point is inside - calls, you can do that with `helm-gtags-tags-in-this-function`, which is bound to C-c g a in my setup.

Find files in project

- Using `ggtags`: Run `ggtags-find-file` to find a file from all the files indexed. If point is on an included header file, `ggtags-find-tag-dwim` automatically jumps to the file.
- Using `helm-gtags`: Run `helm-gtags-find-files` to find files matching regexp. If point is on an included header file, `helm-gtags-dwim` automatically jumps to the file.

Alternatively, you have a more generic solution, that is using Projectile. Projectile is a generic project management tool that you learn later. With Projectile, jumping around version controlled project like Linux kernel is a breeze, since you can jump to any file regardless of where you are standing in the project.

View visited tags with tag stack

- Using `ggtags`: As you know that you can jump back with `pop-tag-mark` (bound to `M-,`), you can also view a list of visited tags using `ggtags-view-tag-history`, which is bound to `C-c g h`. It displays visited tags from newest to oldest, that is from top to bottom.
- Using `helm-gtags`: Similarly, `helm-gtags` also has the command `helm-gtags-show-stack` that shows visited tags from newest to oldest, from top to bottom.

Browse source tree with `Speedbar` file browser

If you want a static outline tree, Emacs also has a more one: `Speedbar`. To use Speed bar, `M-x speedbar` and a frame that contains a directory tree appear. In this directory, to the left of a file or directory name is an icon with `+` sign in it. You can click the icon to open the content of a node. If the node is a file, the children of the files are tags (variable and function definitions) of the file; if the node is a directory, the children of the node are files in that directory. One important thing to remember, Speedbar only lists files that match `speedbar-file-regexp`, that contains the extensions for common programming languages. If you don't see files in your programming languages listed, consider adding it the regexp list.

Basic usage:

- Use SPC to open the children of a node.
- RET to open the node in another window. If node is a file, open that file; if node is a directory, enter that directory; if node is a tag in a file,

Table of Contents

jump to the location of that tag in the file.

- U to go up parent directory.
- n or p moves to next or previous node.
- M-n or M-p moves to next or previous node at the current level.
- b switches to buffer list using Speedbar presentation. You can also open children of each buffer.
- f switches back to file list.

To enable `speedbar` to show all files:

```
(setq speedbar-show-unknown-files t)
```

Package: `sr-speedbar`

However, you may feel that a frame is difficult to use. To solve this issue, you need `sr-speedbar`, which can be installed via MELPA.

- To open `sr-speedbar`, execute the command `sr-speedbar-open` or `sr-speedbar-toggle`.
- To close `sr-speedbar`, execute the command `sr-speedbar-close` or `sr-speedbar-toggle` again.

Best is to use `sr-speedbar-toggle` only, for simplicity.

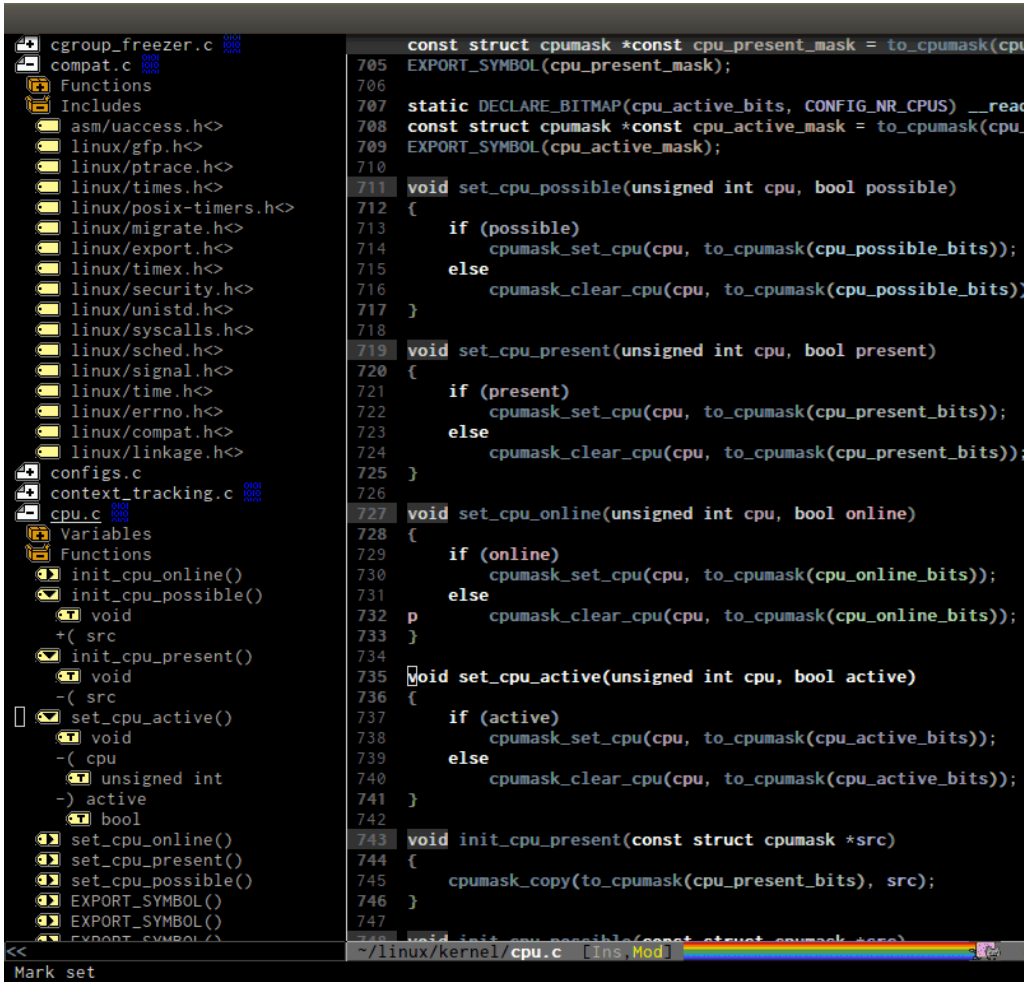
`sr-speedbar` gives the following improvements:

- Automatically switches directory tree - when you switch buffer - to the `default-directory` of current buffer.
- Use an Emacs window instead of frame, make it easier to use.
- C-x 1 deletes every window except Speedbar, so you won't have to open again.
- You can prevent C-x o to jump to `sr-speedbar` window by setting `sr-speedbar-skip-other-window-p` to `t`. You can still move to `sr-speedbar` window using either the mouse or windmove.

Demo: In the demo, you can see that the function `set-cpu-active` is being highlighted. That's what happens when you press RET on a tag: Speedbar moves to the location of that tag and highlight it. Looking at the Speedbar, under `set-cpu-active` node, it contains these children:

- The first child is always the return type, `void`.
- The subsequent children are function parameters. Inside each function parameter node is its type.

Table of Contents



General completion with `company-mode`

`company-mode` is a text completion framework for Emacs. The name stands for "complete anything". It uses pluggable back-ends and front-ends to retrieve and display completion candidates.

It comes with several back-ends such as Elisp, Clang, Semantic, Eclim, Ropemacs, Ispell, CMake, BBDB, Yasnippet, dabbrev, etags, gtags, files,

Table of Contents

keywords and a few others.

After installing `company-mode` from MELPA, activate it globally:

```
(require 'company)
(add-hook 'after-init-hook 'global-company-mode)
```

General Usage: Completion will start automatically after you type a few letters. Use M-n and M-p to select, <return> to complete or <tab> to complete the common part. Search through the completions with C-s, C-r and C-o. Press M-(digit) to quickly complete with one of the first 10 candidates. When the completion candidates are shown, press <f1> to display the documentation for the selected candidate, or C-w to see its source. Not all back-ends support this.

The variable `company-backends` specifies a list of backends that `company-mode` uses to retrieves completion candidates for you.

That's the basic. In the later sections, you will configure `company-mode` to provide completion candidates.

Demo project

I uploaded a demo project for you to play with completion feature here. The project has this structure:

```
project_root/
            Makefile
            src/
                main.c
                lib.c
                lib2.c
                feature1/
                    feature1.c
            include1/
                lib.h
                feature1/
                    feature1.h
            include2/
                lib2.h
```

For the `.h` files, all have this content:

```
void filename_func1();
int filename_func2(int a, int b);
```

Table of Contents

For `.c` files, except for `main.c`, all have this template:

```
#include "filename.h"

void filename_func1() { }

int filename_func2(int a, int b) { }
```

`filename` or `FILENAME` is actual filename like `lib1`, `lib2`...

The files look silly but good enough for our learning purpose.

OPTION 1: Source code completion using Clang

To use `company-mode` with Clang, add this configuration:

```
(setq company-backends (delete 'company-semantic company-backends))
(define-key c-mode-map  [(tab)] 'company-complete)
(define-key c++-mode-map  [(tab)] 'company-complete)
```

You have two commands for completing C/C++ code:

- `company-clang`: use `clang` to retrieve completion candidates. You will have completion with system header files, but not your project. By default, `company-complete` already includes `company-clang` backend, so you can use `company-complete` to complete for many thing. Note that in the configuration for `company-mode` above, we have to delete `company-semantic`, otherwise `company-complete` will use `company-semantic` instead of `company-clang`, because it has higher precedence in `company-backends`. `company-semantic` will be discuss in the CEDET section.

  To retrieve completion candidates for your projects, you will have to tell Clang where your include paths are. Create a file named `.dir-locals.el` at your project root:

  ```
  ((nil . ((company-clang-arguments . ("-I/home/<user>/project_root/include1/"
                                       "-I/home/<user>/project_root/include2/")))))
  ```

If you put a file with a special name .dir-locals.el in a directory, Emacs will read it when it visits any file in that directory or any of its subdirectories, and apply the settings it specifies to the file's buffer. If you want to know more, read GNU Emacs Manual - Per-Directory Local Variables. If you use Helm, you can easily insert absolute path by C-c i at the current path in `helm-find-files` (bound to C-x C-f by default in my demo `.emacs.d` at the beginning):

Table of Contents

```
(
 (nil . ((company-clang-arguments . ("-I"
                                     "-I"))))
 )
```

~/workspace/project_root/.dir-locals.el [Ins,Mod] (First workgroup) (2,37)
Eval: START

In the above example, `nil` means apply the settings to any file or sub-directory visited. If `non-nil`, you have to specify a major mode and the settings are applied to major modes only. You set `company-clang` to `c-mode` and `c++-mode` anyway, so there's no problem setting major mode to

Table of Contents

nil. The remaining is a key-value pair of variable and value of that variable. `company-clang-arguments` is where you can tell include paths, and it takes a list of strings of include paths, as you can see above. After that, `company-clang` can see include paths in your project. If you add something new, like an include path, to your `.dir-locals.el` and is editing some source file, reopen the file for the new values to take effect.

- `company-gtags`: use GTAGS database from `GNU Global` to retrieve completion candidates. It displays ALL completion candidates in GTAGS database regarding of scope. Use `company-gtags` to provide code completion in your current project.

Exercise:

- Create a `.dir-locals.el` at `project_root`.
- Add this Emacs Lisp code:

```
((nil . ((company-clang-arguments . ("-I/home/<user>/project_root/include1/"
                                     "-I/home/<user>/project_root/include2/")))))
```

Replace `<user>` with your username.

- Enter any source file and start completion. You will see that `company-clang` correctly retrieves completion candidates in those directories.

Package: `company-c-headers`

company-c-headers provides auto-completion for C/C++ headers using Company. After installing from MELPA, set it up:

```
(add-to-list 'company-backends 'company-c-headers)
```

Table of Contents

IMPORTANT: If you want to complete C++ header files, you have to add its paths since by default `company-c-headers` only includes these two system include paths: `/usr/include/` and `/usr/local/include/`. To enable C++ header completion for standard libraries, you have to add its path, for example, like this:

```
(add-to-list 'company-c-headers-path-system "/usr/include/c++/4.8/")
```

After that, you can complete C++ header files. To complete project local, use `company-c-headers-path-user` and put it in `.dir-locals.el`.

OPTION 2: Source code completion using CEDET

What is CEDET?

CEDET is a (C)ollection of (E)macs (D)evelopment (E)nvironment (T)ools written with the end goal of creating an advanced development environment in Emacs. CEDET includes common features such as intelligent completion, source code navigation, project management, code generation with templates . CEDET also provides a framework for working with programming languages; support for new programming languages can be added and use CEDET to provide IDE-like features.

Table of Contents

This tutorial only helps you use CEDET to get completion feature for C/C++. Finally, why is this part option? CEDET has a limitation that people don't like: Syntax analyzing takes time. Many people don't understand this process and assume that CEDET is slow. It is not that the fault of CEDET. Because real parsing takes time and such a task, while possible using Emacs Lisp, but is too much for current Emacs implementation. CEDET does some nice optimizations to speed it up as fast as it can: idle parsing, caching and incremental parsing to reduce computational time. Consider a project called clang-ctags using Clang to generate tag database:

> "Running clang-ctags over a much larger input, such as the entire llvm C/C++ sources (7k files, 1.8 million lines of code) took 98 minutes and a peak memory usage of 140MB."

The more accurate to analyze the source code, the more time the parser needs to spend. If you don't like to wait, skip this section and use `company-clang` instead. At least you will have completion with system header files, but not your project.

Why use CEDET?

CEDET is simple to setup and portable (right winthin Emacs and written entirely with Emacs Lisp). Without CEDET, you have to use external tools and third party Emacs packages. The downside of external tools is that they are not specifically designed for Emacs. They have similar use cases, but not always satisfying. For example, source code indexing tools such as GNU Global and Exuberant Ctags are really good at working static source code, but they do not keep track changes in real time; CEDET does:

Table of Contents

```
1 #include <algorithm>
2 #include <string>
3 #include <iostream>
4 #include <map>
5 #include "coloring_solver.h"
6 #include <stdio.h>
7
8 Answer ColoringSolver::solve() {
9     using namespace std;
10
11     p
12
13     set<int> used_colors, viable_colors;
14     map<int, set<int> > original;
15
16     for (auto& u:vertices)
17     {
18         for (auto& v:u.end_vertices)
19         {
20             original[u.id].insert(v.second->id);
21         }
22     }
23
24     // for (auto& o : original)
25     // {
26     //     cout << "origin i: " << o.first << endl;
27     //     cout << "origin end vertices: ";
28     //     for (auto& id : o.second )
29     //     {
30     //         cout << id << ' ';
31     //     }
32     //     cout << endl;
33     // }
34
35     sort(vertices.begin(), vertices.end(), [] (const Vertex& a, const Vertex& b) {
36             return a.end_vertices.size() > b.end_vertices.size();
37         });
38
39     for (auto& u : vertices )
40     {
41         for (auto& v_idx:original[u.id])
42         {
43             auto it = find_if (vertices.begin(), vertices.end(), [&v_idx] (const Vertex& o) -> bool {
44                     return o.id == v_idx;
.../discrete_optimization/hw2/coloring/coloring_solver.cpp GG [Ins]  (11, 5) [Top/9.4k] [C++/l] [Git:mas
call to non-static member function without an object argument
```

As you can see, CEDET recognizes when `printk.h` is included and provides appropriate completion candidates. In contrast, non-context sentisve completion is like this:

Table of Contents

```
1  #include "coloring_solver.h"
2  #include <algorithm>
3  #include <string>
4  #include <iostream>
5  #include <map>
6  #include "coloring_solver.h"
7
8  Answer ColoringSolver::solve() {
9      using namespace std;
10
11     set<int> used_colors, viable_colors;
12     map<int, set<int> > original;
13
14     pr
15
16     for (auto& u:vertices)
17     {
18         for (auto& v:u.end_vertices)
19         {
20             original[u.id].insert(v.second->id);
21         }
22     }
23
24     // for (auto& o : original)
25     // {
26     //     cout << "origin i: " << o.first << endl;
27     //     cout << "origin end vertices: ";
28     //     for (auto& id : o.second )
29     //     {
30     //         cout << id << ' ';
31     //     }
32     //     cout << endl;
33     // }
34     sort(vertices.begin(), vertices.end(), [] (const Vertex& a, const Vertex& b) {
35             return a.end_vertices.size() > b.end_vertices.size();
           });
37
38     for (auto& u : vertices )
39     {
40         for (auto& v_idx:original[u.id])
41         {
42             auto it = find_if (vertices.begin(), vertices.end(), [&v_idx] (const Ve
43                     return o.id == v_idx;
.../discrete_optimization/hw2/coloring/coloring_solver.cpp  [Ins,Mod] (First workgroup
Eval: START
```

In this case, the completion system gets all candidates straight from GNU Global generated database without considering current context.

The disadvantage is that CEDET is written in Emacs Lisp, and it is bound to the performance limitations of Emacs. Even though, CEDET is still really fast for moderate-size source files. CEDET also makes use of external tools like GNU Global or Cscope or Exuberant Ctags for finding symbol references/definition in a project to reduce its workload.

Table of Contents

Installation

CEDET was merged into Emacs since 23.2. You do not need to install CEDET manually. However, you can also use the development repository that contains latest bug fixes and more features. Nevertheless, the built-in Emacs is still adequate for daily usage and convenient for trying out before actually cloning and use the development version. Skip this section if you only want to try Emacs. Come back later if you really like it.

If you really want to use the development version with latest feature, checkout this branch:

```
git clone http://git.code.sf.net/p/cedet/git cedet
```

Be sure to place the checked out `cedet` directory in your `~/.emacs.d`. Then compile it:

```
cd cedet
make # wait for it to complete
cd contrib
make
```

Finally, assume that you placed your newly cloned CEDET in `~/.emacs.d`, load it into your Emacs:

```
(load-file (concat user-emacs-directory "/cedet/cedet-devel-load.el"))
(load-file (concat user-emacs-directory "cedet/contrib/cedet-contrib-load.el"))
```

Semantic minor modes

`Semantic` is a package that provides language-aware editing commands based on source code parsers. Parsing is a process of analyzing source code based on programming language syntax. Emacs understands your source code through this process to provides features such as contextual code completion, code navigation. Here is an example how Semantic helps you provides completion for Boost:

Table of Contents

```
1  #include <boost/asio.hpp>
2
3  int main(int argc, char *argv[])
4  {
5      
6      return 0;
7  }
~/test.cpp  [Ins,Mod] (test)          ( 5, 4) [All/83] [C++/l] company
Eval: START
```

Table of Contents

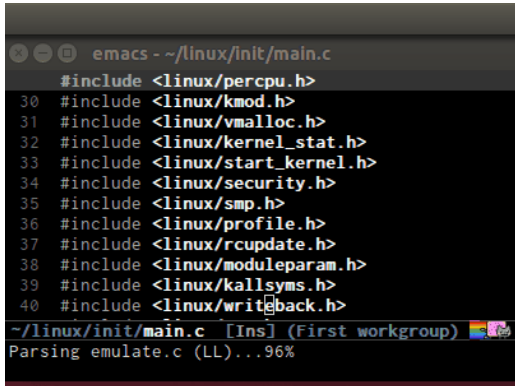
- Setup Semantic

  To enable code completion using Semantic, add the following code:

```
(require 'cc-mode)
(require 'semantic)

(global-semanticdb-minor-mode 1)
(global-semantic-idle-scheduler-mode 1)

(semantic-mode 1)
```

- Command: `semantic-mode`

  This command activates `semantic-mode`. In Semantic mode, Emacs parses the buffers you visit for their semantic content. The parsing looks like this:



  Notice that at the bottom, Emacs prints messages like this: `Parsing <file> (LL)...<progress in percentage>`. This is common behavior of `Semantic`: when you perform a jump to a symbol or perform a completion, `Semantic` tries to be accurate by parsing all the included files and all the included files in the included files and so on, until it reaches the end:

  You may worry that this process takes a long time. Sometimes, it does. However, this is just a one time process. Once it is parsed, Semantic will cache the parsing result for future use. The next time you perform some action on the parsed code, such as code completion, you will get your desired output instantly.

- Command: `global-semanticdb-minor-mode`

Table of Contents

As mentioned above, Semantic caches parsing result for future use. To do that, `semanticdb-minor-mode` must be activated. In Semantic DB mode, Semantic parsers store results in a database, which can be saved for future Emacs sessions. The cache is saved in directory specified by `semanticdb-default-save-directory` variable. The default directory is `~/.emacs.d/semanticdb`.

- Command: `global-semantic-idle-scheduler-mode`

  When `semantic-idle-scheduler-mode` is enabled, Emacs periodically checks to see if the buffer is out of date, and reparses while the user is idle (not typing). When this mode is off, a buffer is only reparsed when user explicitly issue some command.

  With `semantic-idle-scheduler-mode`, Emacs keeps track live changes of your source code.

- Add more system include paths

  By default, Semantic automatically includes some default system include paths such as `/usr/include`, `/usr/local/include`... You can view the list of include paths in `semantic-dependency-system-include-path`. To add more include paths, for example Boost include paths, use the function `semantic-add-system-include` like this:

```
(semantic-add-system-include "/usr/include/boost" 'c++-mode)
(semantic-add-system-include "~/linux/kernel")
(semantic-add-system-include "~/linux/include")
```

  If you want the system include paths to be available on both C/C++ modes, then ignore the optional mode argument in `semantic-add-system-include`.

- Completion using `company-mode`

  `companh-mode` provides a command called `company-semantic` that uses SemanticDB to retrieve completion candidates. Function interface of each candidate is shown in the minibuffer. One nice thing of `company-semantic` is that it fixed an issue of original Semantic completion `semantic-ia-complete-symbol`: it can show you completions even if there's no prefix. The original `semantic-ia-complete-symbol` requires to have at least one character as a prefix for finding completions.

- Package: `function-args`

  function-args is a GNU Emacs package for showing an inline arguments hint for the C/C++ function at point.

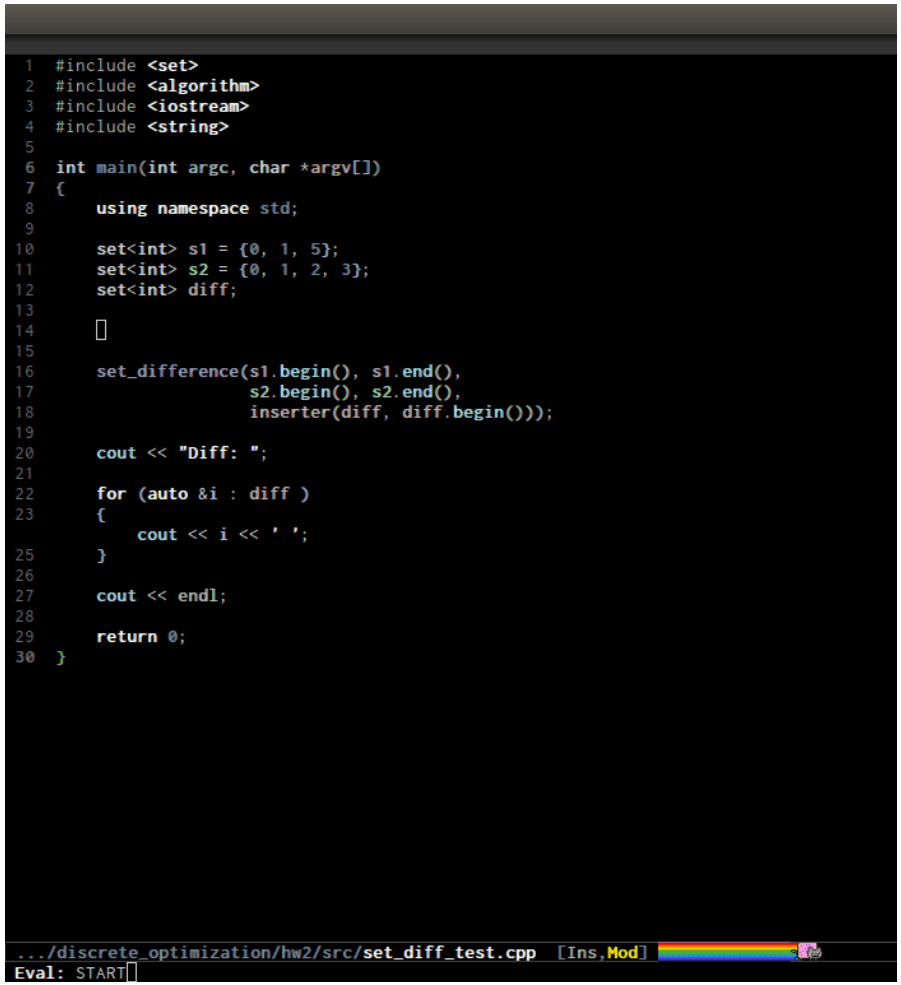
  Setup:

```
(require 'function-args)
(fa-config-default)
(define-key c-mode-map  [(control tab)] 'moo-complete)
(define-key c++-mode-map  [(control tab)] 'moo-complete)
(define-key c-mode-map (kbd "M-o")  'fa-show)
(define-key c++-mode-map (kbd "M-o")  'fa-show)
```

Table of Contents

Basic Usage (taken from `function-args` homepage):

- `fa-show`: Show an overlay hint with current function arguments like so:

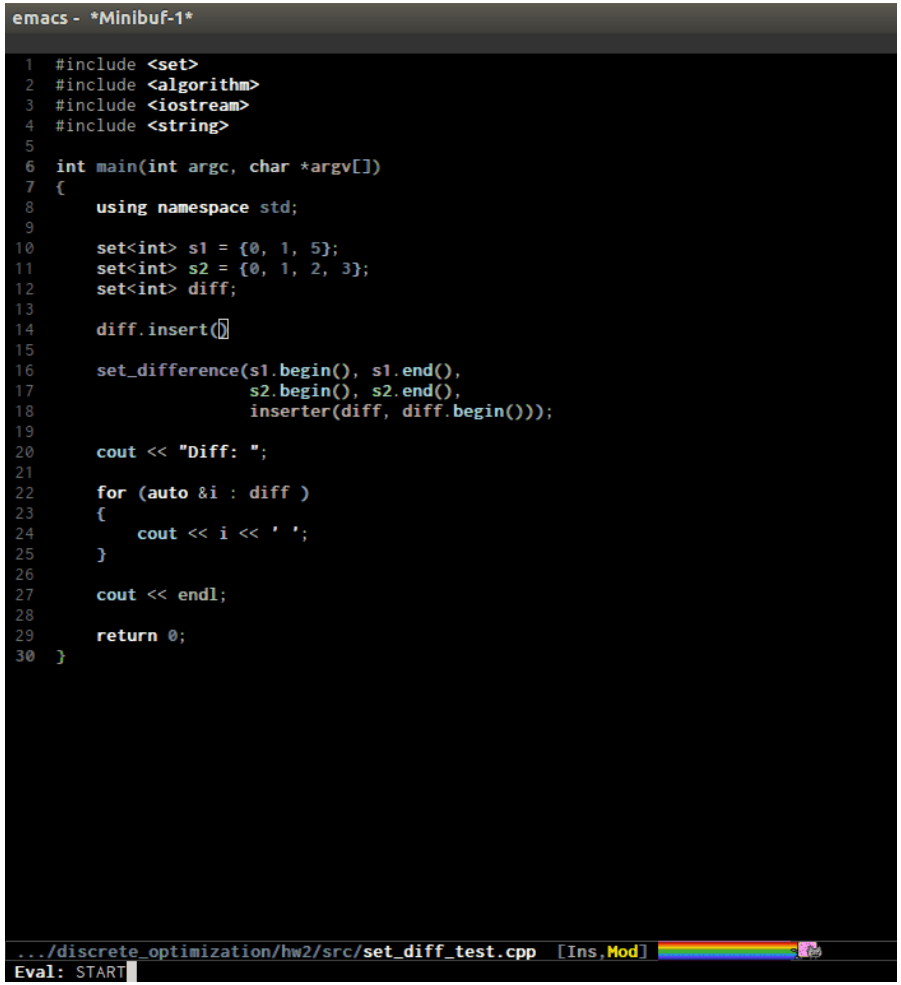


The point position is tracked and the current hint argument is updated accordingly. After you've called it with `M-o`, you can cycle the overloaded functions with `M-n/M-h`. You can dismiss the hint with `M-u` or by editing anywhere outside the function arguments.

- `fa-jump`: While the overlay hint from `fa-show` is active, jump to the current function. The default shortcut is `M-j`. If the overlay

Table of Contents

isn't active, call whatever was bound to `M-j` before (usually it's `c-indent-new-comment-line`). Demo:



- `moo-complete` (must install `helm`): It's essentially a c++-specific version of semantic-ia-complete-symbol. It behaves better, because it accounts more for function overloading and inheritance. Also it's prettier (type parts are fontified) and faster (helm is used for completion). You can invoke it with `M-o` by default.

There are a few more commands. You can view all the descriptions for all commands here.

Table of Contents

CEDET can do more

This guide only covers a portion of CEDET. Since Semantic parses source code and creates a database for code completion, it would be useful to reuse the database for other features such as code navigation, jump to definition and gather references. These are cool features. For example, CEDET provide nice syntax highlighting for gathering references:

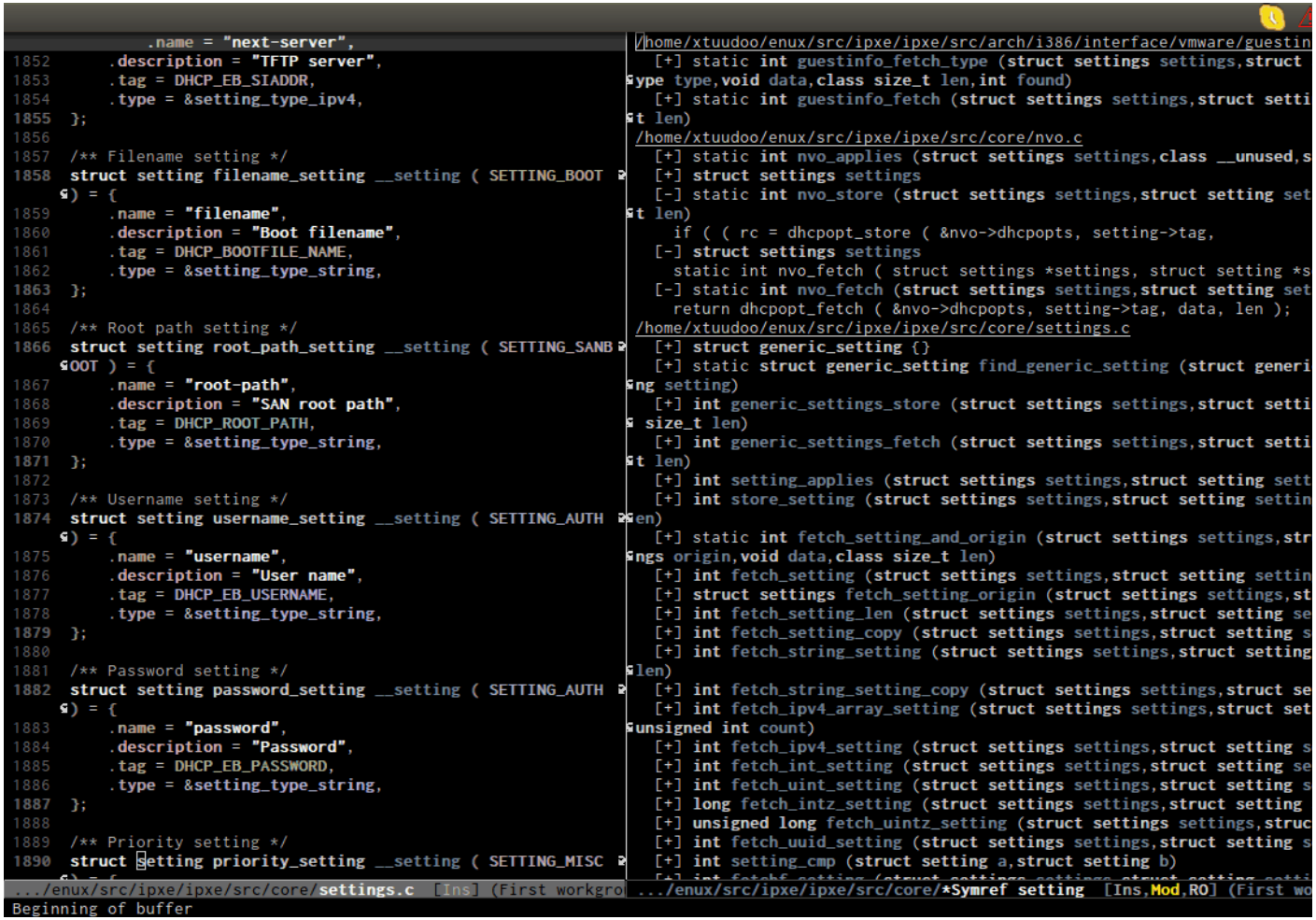


Table of Contents

The above feature is called Semantic Symref. It queries tag references from SemanticDB and display the result in another buffer. Please refer to GNU Manual - Symbol References for more info. If your project is only the size of Emacs or similar, then Semantic Symref is a viable choice. Remember that when entering new files, Semantic takes time to parse and if you gather references for the first time, you will have to wait for a while for Semantic doing its job. After the first time, subsequent uses of Semantic Symref happens instantly.

But, for navigating around the source tree, we already have `ggtags` and `helm-gtags`, which uses GNU Global for much faster indexing for large project like Linux kernel. The only thing that other tools cannot do, is context-sensitive completion. That's why we only use CEDET for code completion in this guide. Nevertheless, Semantic Symref is still a viable choice for small to medium sized projects. Choose the one you prefer.

Other solutions that use `clang` is quite good but not there yet. For example, auto-complete-clang is fine for getting system header candidates (since `clang` has system paths by default), but it has no concept of project and is not aware of project include path. You can add more arbitrary include paths to auto-complete-clang, but it won't be pretty: once you add an include path for a project, it is visible to all other projects since the include path is treat as system include path. That means, whenever you try to complete something, you get irrelevant candidates from other projects as well. Quite annoying. `company-clang` also has the same problem. Another solution is rtags, but it is really complicated to setup, especially if you use `make`; I never succeed with it. `clang` based packages still have a long way to go. CEDET also supports `clang` for retrieving completion candidates. It also has the limitations of other packages.

CEDET is best used with new project, because Semantic parse code as you write. As a result, you won't have to wait for parsing unknown source files to get completion candidates.

Project management with EDE

EDE, short for Emacs Development Environment, is a generic interface for managing projects. In EDE, a project hierarchy matches a directory hierarchy. The project's topmost directory is called the project root, and its subdirectories are sub-projects.

EDE can do many things but we will just use it for code completion at project level. To demonstrate the use of EDE, we will create a little project of our own.

To setup EDE:

```
(require 'ede)
(global-ede-mode)
```

Now, let's try completion in `main.c` using `moo-complete` or `company-semantic`. Nothing also happens. It is because Semantic only looks for header files in current directory by default. If you put it elsewhere, you have to tell Semantic where it is. This is not limited to only Semantic; you have to specify project include path in Eclipse as well.

- Create a new file called `cedet-projects.el` in `~/.emacs.d/`.
- In this file, add a new `ede-cpp-root-project` in this form:

```
(ede-cpp-root-project "project_root"
```

Table of Contents

```
                        :file "/dir/to/project_root/Makefile")
```

The first argument to `ede-cpp-root-project` is project name. `:file` argument specifies path to project root. You must create a file in the project root, since EDE uses that file as an "anchor" to project root; in our case, `Makefile` is the anchor file. Not sure why EDE just straightly uses root directory.

- Add include directories specific to the project and in your system:

```
(ede-cpp-root-project "project_root"
                      :file "/dir/to/project_root/Makefile"
                      :include-path '("/include1"
                                      "/include2") ;; add more include
                      ;; paths here
                      :system-include-path '("~/linux"))
```

`:include-path` specifies directories local to your projects that EDE should search first when looking for a header file. `:incluide-path` is relative to project root specified in `:file`.

`:system-include-path` specifies system include paths that do not belong to current project. Note that despite it is called `system-include-path`, it does not have to be in place like `/usr/include`. You can specify any include directories outside of your current project as "system headers".

After done setting up your project, save the file and execute that `ede-cpp-root-project` expression with C-x C-e. Now, Semantic is aware of your project include paths. However, you have to do one last thing: either close the file `main.c` and reopen it or `M-x semantic-force-refresh` to tell Semantic to analyze `main.c` again. Otherwise, Semantic will still keep the previous parsing result of this file and completion candidates won't be available. As a result, it is important to load EDE projects defined by `ede-cpp-root-project` before opening any project file.

After that, try auto-completion and you will see everything is working fine again:

Table of Contents

```
1  #include "lib1.h"
2  #include "lib2.h"
3
4  #include "feature1/feature1.h"
5
6  int main(int argc, char *argv[])
7  {
8      
9
10     return 0;
11 }
```

```
/home/tuhdo/workspace/project_root:
.
..
? include1
? include2
? src
? Makefile

/home/tuhdo/workspace/project_root/include1:
.
..
feature1
lib1.h

/home/tuhdo/workspace/project_root/include1/feature1:
.
..
feature1.h

/home/tuhdo/workspace/project_root/include2:
.
..
lib2.h
lib3.h

/home/tuhdo/workspace/project_root/src:
.
..
feature1.c
lib1.c
lib2.c
main.c
```

```
.../workspace/project_root/src/main.c  [Ins,Mod] (test)
```

```
~/workspace/project_root/project_root  [Ins,Mod,RO] (test)
```

```
1  (ede-cpp-root-project "project_root"
2                        :file "~/workspace/project_root/Makefile"
3                        :include-path '("/include1"
4                                        "/include2") ;; add more include paths here
5                        :system-include-path '("~/linux"))
```

```
~/project-cedet.el  [Ins] (test)  ( 5,56) [All/290] [Emacs-Lisp] company
Eval: START
```

Table of Contents

Summary:

- EDE, short for Emacs Development Environment, is a generic interface for managing projects.
- EDE enables Semantic to find sources and perform project level completions.
- To add a C/C++ project, simply create a project definition with this template in a file, such as `ede-projects.el`:

```
(ede-cpp-root-project "project_root"
                      :file "/dir/to/project_root/Makefile"
                      :include-path '("/include1"
                                      "/include2") ;; add more include
                      ;; paths here
                      :system-include-path '("~/linux"))
```

- Load `ede-projects.el` when Emacs start.
- If you have opening files, either close or refresh it with `M-x semantic-force-refresh` to make Semantic aware of new changes.

Later, you will also learn another project management tool called `Projectile`. You may ask, why another project management tool? The differences are:

- EDE is older, Projectile appears recently.
- EDE manages project files to integrate with specific build system and generate corresponding build file (such as Makefile; these features are not covered in this guide). Projectile provides generic file management operations such as quickly jump to file/directory in project, list buffers that belong to current project...
- EDE is part of CEDET suite; it is geared toward Semantic. For example, we used EDE to tell Semantic where to retrieve correct include paths for current project to perform smart completion in previous section. Projectile provides a generic interface for managing your project under a VCS or some supported build system.
- Projectile is easier to learn than EDE.

Both have some overlapping in features, such as Projectile provides basic tag navigation in project, but in general they support each other. For our basic usage, we use EDE for smart completion as in previous section and Projectile to navigate our project effortlessly.

Source code navigation using Senator

Senator is a part of CEDET. Senator stands for SEmantic NAvigaTOR. Senator provides some useful commands for operating on semantic tags in SemanticDB. As you can see, another utility makes use of SemanticDB, aside from smart completion. It is like the heart of CEDET: once Semantic fails to parse, tools centered around it fail as well. If such situation happens, you always have a reserved and simpler solution: use GNU Global with `ggtags` or `helm-gtags` frontends. The following commands are provided by Senator:

Navigation

Table of Contents

Senator provides commands for navigating by tag.

- C-c , n runs `senator-next-tag`, navigate to the next Semantic tag.
- C-c , p runs `senator-previous-tag`, navigate to the previous Semantic tag.

Here is how both of those commands work:

Table of Contents

```
static int __init rdinit_setup(char *str)
    for (i = 1; i < MAX_INIT_ARGS; i++)
        argv_init[i] = NULL;
    return 1;
}
__setup("rdinit=", rdinit_setup);

#ifndef CONFIG_SMP
static const unsigned int setup_max_cpus = NR_CPUS;
#ifdef CONFIG_X86_LOCAL_APIC
static void __init smp_init(void)
{
    APIC_init_uniprocessor();
}
#else
#define smp_init()  do { } while (0)
#endif

static inline void setup_nr_cpu_ids(void) { }
static inline void smp_prepare_cpus(unsigned int maxcpus) { }
#endif

/*
 * We need to store the untouched command line for future reference.
 * We also need to store the touched command line since the parameter
 * parsing is performed in place, and we should allow a component to
 * store reference of name/value for future reference.
 */
static void __init setup_command_line(char *command_line)
{
    saved_command_line =
        memblock_virt_alloc(strlen(boot_command_line) + 1, 0);
    initcall_command_line =
        memblock_virt_alloc(strlen(boot_command_line) + 1, 0);
    static_command_line = memblock_virt_alloc(strlen(command_line) + 1, 0);
    strcpy (saved_command_line, boot_command_line);
    strcpy (static_command_line, command_line);
}

/*
 * We need to finalize in a non-__init function or else race conditions
 * between the root thread and the init thread may cause start_kernel to
 * be reaped by free_initmem before the root thread has proceeded to
 * cpu_idle.
 *
 * gcc-3.4 accidentally inlines this function, so use noinline.
 */

static __initdata DECLARE_COMPLETION(kthreadd_done);

static noinline void __init_refok rest_init(void)
{
    int pid;
~/linux/init/main.c  [Ins] (test)                    (350, 0) [36%/23k]
Eval: START
```

Table of Contents

As you see,the C macro (in the screenshot, its `DECLARE_COMPLETION`) is not considered a tag.

- `senator-jump` (only available in CEDET bzr), specify a tag to jump to. TAB for a list of available Semantic tags in current buffer, similar to TAB in M-x in stock Emacs. If you use Helm, use `moo-jump-local` from the package `function-args` instead.

  Demo (`helm-mode` is disabled):

Table of Contents

```
 1  /*
 2   *  linux/init/main.c
 3   *
 4   *  Copyright (C) 1991, 1992  Linus Torvalds
 5   *
 6   *  GK 2/5/95  -  Changed to support mounting root fs via NFS
 7   *  Added initrd & change_root: Werner Almesberger & Hans Lermen, Feb '96
 8   *  Moan early if gcc is old, avoiding bogus kernels - Paul Gortmaker, May '96
 9   *  Simplified starting of init:  Michael A. Griffith <grif@acm.org>
10   */
11
12  #define DEBUG       /* Enable initcall_debug */
13
14  #include <linux/types.h>
15 !#include <linux/module.h>
16  #include <linux/proc_fs.h>
17  #include <linux/kernel.h>
18  #include <linux/syscalls.h>
19  #include <linux/stackprotector.h>
20  #include <linux/string.h>
21  #include <linux/ctype.h>
22  #include <linux/delay.h>
23  #include <linux/ioport.h>
24  #include <linux/init.h>
25  #include <linux/initrd.h>
26  #include <linux/bootmem.h>
    #include <linux/acpi.h>
    #include <linux/tty.h>
29  #include <linux/percpu.h>
30  #include <linux/kmod.h>
31  #include <linux/vmalloc.h>
32  #include <linux/kernel_stat.h>
33  #include <linux/start_kernel.h>
34  #include <linux/security.h>
35  #include <linux/smp.h>
36  #include <linux/profile.h>
37  #include <linux/rcupdate.h>
38  #include <linux/moduleparam.h>
39  #include <linux/kallsyms.h>
40  #include <linux/writeback.h>
41  #include <linux/cpu.h>
42  #include <linux/cpuset.h>
43  #include <linux/cgroup.h>
44  #include <linux/efi.h>
45  #include <linux/tick.h>
46  #include <linux/interrupt.h>
47  #include <linux/taskstats_kern.h>
48  #include <linux/delayacct.h>
49  #include <linux/unistd.h>
50  #include <linux/rmap.h>
51  #include <linux/mempolicy.h>
52  #include <linux/key.h>
~/linux/init/main.c  [Ins] (test)                    ( 1, 0) [Top/23k] [C/l] company
Eval: START
```

- C-c , u runs `senator-go-to-up-reference`, move up one reference level from current tag. An upper reference level of a tag is the

source that defines the tag or includes the tag. This is incredibly useful when you want to jump from a function declaration in a class to its definition, or jump to the class that a function belongs to, quickly.

Table of Contents

Table of Contents

```
1  #ifndef PIC32F42_H
2  #define PIC32F42_H
3  #include "MicroController.h"
4  using namespace std;
5  class PIC32F42:public MicroController
6  {
7  private:
8      unsigned char W;
9      short GetAddressToOperate(short Opcode);
10 public:
11     PIC32F42(void);
12     PIC32F42(string Name, short MemorySize);
13     virtual ~PIC32F42(void);
14     void Reset(void);
15     virtual bool Execute(void);
16     virtual vector<ERR_MSG> Execute(short Address);
17
18     /* The first byte after the opcode is the value to be written to the W register.
19      * The PC is set to the address of the second byte after the opcode.
20      */
21     ErrorReport MoveValueToW();
22
23     /* The first byte after the opcode is the high byte of the memory address,
24      * and the second byte is the low byte of the memory address.
25      * The memory address can be determined by (high byte << 8) | low byte.
26      * The W register"s contents are written to this memory address. The
27      * PC is set to the address of the third byte after the opcode.
28      */
29     ErrorReport MoveWToMemory();
30
31     /* The first byte after the opcode is the value to be added to the W register.
32      * The PC is set to the address of the second byte after the opcode.
33      */
34     ErrorReport AddValueToW();
35
36     /* The first byte after the opcode is the value to be subtracted from the W register.
37      * The PC is set to the address of the second byte after the opcode.
38      */
39     ErrorReport SubtractValueFromW();
40
41     /* The first byte after the opcode is the high byte of the address.
42      * The second block of memory after the opcode is the low byte of the address.
43      * After execution, the PC points to that address.
44      */
45     ErrorReport GoToAddress();
46
47     /* The next value after the opcode is the comparison value.
48      * The second block of memory after the opcode is the high byte of the branch target,
49      * and the third block of memory is the low byte of the branch target.
50      * If the W register is not the same as the comparison value, then the
51      * program counter is set to equal the branch target. Otherwise, the program
52      * counter is set to equal the fourth block of memory after the opcode.
.../s3183504_CPP_A1_BarendScholtus/PIC32F42.h  [Ins,Mod] (test)                (1
Eval: START
```

Table of Contents

In the demo, I only use `senator-go-to-up-reference` to switch from a declaration, that is `PIC32F42(string Name, short MemorySize);` in `PIC32F42.h` to its implementation in `PIC32F42.c`. C-c , u again, point jump is on the class `PIC32F42`, which is the class that the function belongs. Class `PIC32F42` has its parent class `Microcontrller`, and I run C-c , u again to move point to `Microcontrller` class. At this point, I cannot move further.

Semantic also provides a useful command for finding all references of a tag, that is `semantic-symref`, as demonstrated in previous section. On a symbol, C-c , g and a prompt appear asking for a tag for gathering references, with the default is the symbol at point. Leave prompt blank and RET to use the default or enter another symbol if you change your mind.

Copy/Paste

- `C-c , M-w` runs `senator-copy-tag`, take the current tag, and place it in the tag ring.
- `C-c , C-w` runs `senator-kill-tag`, take the current tag, place it in the tag ring, and kill it. Killing the tag removes the text for that tag, and places it into the kill ring. Retrieve that text with C-y.

  The above commands are basically like normal `M-w` and `C-w`, but are used for tags. For example, run `C-c , C-w` with point inside a function definition kills the whole function body and its interface, or you can kill function parameters, and can be yanked back with C-y:

Table of Contents

```
#include "PIC32F42.h"
#include <string>
using namespace std;
PIC32F42::PIC32F42(void)
{
}
/* Since there's no sample for input file, these are the values provided
 * upon the PIC32F42 is created to test.
 */
PIC32F42::PIC32F42(string Name, short MemorySize):MicroController(Name, MemorySize)
{
    /*this->Memory[0x00] = 0x50;
      this->Memory[0x01] = 0xab;

      this->Memory[0x02] = 0x51;
      this->Memory[0x03] = 0x02;
      this->Memory[0x04] = 0x05;

      this->Memory[0x06] = 0x5a;
      this->Memory[0x07] = 0xab;

      this->Memory[0x09] = 0x6e;
      this->Memory[0x0a] = 0x03;
      this->Memory[0x0b] = 0xff;

      this->Memory[0x10] = 0x5B;
      this->Memory[0x11] = 0x11;

      this->Memory[0x12] = 0x70;
      this->Memory[0x13] = 0x22;
      this->Memory[0x14] = 0x02;
      this->Memory[0x15] = 0x0f;    */

    this->W = 0;
    this->ProgramCounter = 0;
}

PIC32F42::~PIC32F42(void)
{
}

void PIC32F42::Reset(void)
{
    ProgramCounter = 0;
    W = 0;
}

bool PIC32F42::Execute(void)
{
    return true;
}

.../s3183504_CPP_A1_BarendScholtus/PIC32F42.cpp  [Ins,Mod] (test)
Eval: START
```

Table of Contents

As you see, the function body and its interface is killed and yanked back with C-y. You can also see that the function parameter, aside from its name, its type is also removed when the command is executed.

- `C-c , C-y` runs `senator-yank-tag`, yank a tag from the tag ring. The form the tag takes is different depending on where it is being yanked to. For example, in previous example with `C-c , C-w`, when you run C-y, it yanks the whole thing including function interface and its body. C-c , C-y only yanks the function interface.

Table of Contents

```
1  #include "PIC32F42.h"
2  #include <string>
3  using namespace std;
4  PIC32F42::PIC32F42(void)
5  {
6  }
7  /* Since there's no sample for input file, these are the values provided
8   * upon the PIC32F42 is created to test.
9   */
10 PIC32F42::PIC32F42(string Name, short MemorySize):MicroController(Name, MemorySize)
11 {
12     /*this->Memory[0x00] = 0x50;
13       this->Memory[0x01] = 0xab;
14
15       this->Memory[0x02] = 0x51;
16       this->Memory[0x03] = 0x02;
17       this->Memory[0x04] = 0x05;
18
19       this->Memory[0x06] = 0x5a;
20       this->Memory[0x07] = 0xab;
21
22       this->Memory[0x09] = 0x6e;
23       this->Memory[0x0a] = 0x03;
24       this->Memory[0x0b] = 0xff;
25
26       this->Memory[0x10] = 0x5B;
27       this->Memory[0x11] = 0x11;
28
29       this->Memory[0x12] = 0x70;
30       this->Memory[0x13] = 0x22;
31       this->Memory[0x14] = 0x02;
32       this->Memory[0x15] = 0x0f;    */
33
34     this->W = 0;
35     this->ProgramCounter = 0;
36 }
37
38 PIC32F42::~PIC32F42(void)
39 {
40 }
41
42 void PIC32F42::Reset(void)
43 {
44     ProgramCounter = 0;
45     W = 0;
46 }
47
48 bool PIC32F42::Execute(void)
49 {
50     return true;
51 }
52
.../s3183504_CPP_A1_BarendScholtus/PIC32F42.cpp  [Ins,Mod] (test)
Eval: START
```

- `C-c , r` runs `senator-copy-tag-to-register`, copy the current tag into a register. With prefix argument will delete the text of the

tag to the kill ring.

Table of Contents

Obsolete commands

Senator also provides commands for code completion: `senator-complete-symbol` and `senator-completion-menu-popup`. Code completion provided by Senator is simpler than the actual completion provided by `semantic-ia-complete-symbol` command that is used by `moo-complete` for a list of candidates with full information, such as complete function interface is displayed correctly. These two completion commands are provided.

Package: `company-c-headers`

company-c-headers provides auto-completion for C/C++ headers using Company. After installing from MELPA, set it up:

```
(add-to-list 'company-backends 'company-c-headers)
```

IMPORTANT: If you want to complete C++ header files, you have to add its paths since by default `company-c-headers` only includes these two system include paths: `/usr/include/` and `/usr/local/include/`. To enable C++ header completion for standard libraries, you have to add its

Table of Contents

path, for example, like this:

```
(add-to-list 'company-c-headers-path-system "/usr/include/c++/4.8/")
```

After that, you can complete C++ header files. To complete project local, use `company-c-headers-path-user` and put it in `.dir-locals.el`.

Navigate system include path

Up until now we were only navigating code in a project using `GNU Global`. How about jumping to system include headers? You have a few options that I know:

Using `Semantic` with `semantic-ia-fast-jump` command

Semantic provides a jump command based on the parsed tags produced by the Semantic C/C++ parser. To jump to any code in system include path, you must first tell Semantic where it is:

```
(semantic-add-system-include "/usr/local/include")
(semantic-add-system-include "~/linux/include")
```

Then, if point is on an include header like `#include <iostream>` or a tag inside `iostream`, executing the command `semantic-ia-fast-jump` will jump to into header file or jump to the tag at point earlier that is inside `iostream`. If the tag is actually within `iostream` but you do not include `iostream`, Semantic won't be able to jump because it sees no appropriate header that contains the tag.

The function `semantic-add-system-include` will add your include paths to `semantic-c-dependency-system-include-path` variable. By default, it only contains `/usr/include`. You can add more such as `/usr/local/include` or your own project, such as `~/linux/include`.

The disadvantage of using Semantic is that for it may take brief while to parse, but for once; the next time you can jump to parsed source instantly.

Using generated database from `GNU Global`

`GNU Global` has an environment variable named `GTAGSLIBPATH`. This variable holds GTAGS database of external libraries that your project depends on but not inside your project. For example, your project may rely on system headers such as `stdio.h`, `stdlib.h`... but these headers are internal to your project. However, remember that you can only jump to tag definitions of external dependencies, and nothing else (such as files or references). But, again, once you are inside the external library, you can start jumping around sicne it becomes your current project.

To make `GNU Global` sees your system headers, follow these steps:

- Export this environment variable in your shell init file, such as `.bashrc` or `.zshrc`:

```
export GTAGSLIBPATH=$HOME/.gtags/
```

Table of Contents

- Execute these commands in your terminal:

```
# Create a directory for holding database, since
# you cannot create a database in your system paths
mkdir ~/.gtags

# Create symbolic links to your external libraries
ln -s /usr/include usr-include
ln -s /usr/local/include/ usr-local-include

# Generate GNU Global database
gtags -c
```

The `-c` option tells GNU Global to generate tag database in compact format. It is necessary because if your project contains C++ headers like `Boost`, without `-c` your GTAGS database can be more than 1 GB. Same goes for `ctags`. The `GNU Global` devs explained that it is because the GTAGS database includes the image of tagged line, and the `Boost` headers have a lot of very long lines.

After all the above steps, restart with a shell loaded with that variable. To verify Emacs gets the variable, `M-x getenv` and enter `GTAGSLIBPATH` and see if your predefined value is available. Executing `ggtags-find-tag-dwim` or `helm-gtags-dwim` jumps to the definition of a system tag like a normal tag.

The disadvantage of using `GNU Global` is that currently it cannot include files without extension. In the C++ system include directory like `/usr/include/c++/4.8/`, it contains files without extension such as `iostream`, `string`, `set`, `map`.... so you can write `#include` directives without having to append `.h` at the end. `GNU Global` devs are considering to add support for this use case.

Project management with Projectile

`Projectile` is a project interaction library for Emacs. Its goal is to provide a nice set of features operating on a project level without introducing external dependencies(when feasible). For instance - finding project files has a portable implementation written in pure Emacs Lisp without the use of GNU find (but for performance sake an indexing mechanism backed by external commands exists as well).

Projectile tries to be practical - portability is great, but if some external tools could speed up some task substantially and the tools are available, Projectile will leverage them.

By default, `git`, `mercurial`, `darcs` and `bazaar` are considered projects. So are `lein`, `maven`, `sbt`, `scons`, `rebar` and `bundler`. If you want to mark a folder manually as a project just create an empty `.projectile` file in it.

You also install `Projectile` using MELPA and setup:

```
(projectile-global-mode)
```

Table of Contents

Prefix key of Projectile is C-c p. Some notable features:

- Jump to any file in the project: C-c p f.
- Jump to any directory in the project: C-c p d.
- List buffers local to current project: C-c p b.
- Jump to recently visited files in project: C-c p e.
- Grep in project: C-c p g s
- Multi-occur in project buffers: C-c p o.
- Simple refactoring with text replace in current project: C-c p r.
- Switch visited projects (visited once an Projectile remembers): C-c p p.

Useful commands for working with C/C++ projects:

- Run compilation command at project root: C-c p c. By default, Projectile prompts the `make` command.
- Switch between `.h` and `.c` or `.cpp`: C-c p a. If the filename - without file extension - of current editing buffer is part of other files, those files are listed as well. If there is only one file with the same name but different extension, switch immediately. Here is a demo:

Table of Contents

```
1  #ifndef __ASM_ALPHA_FPU_H
2  #define __ASM_ALPHA_FPU_H
3
4  #include <asm/special_insns.h>
5  #include <uapi/asm/fpu.h>
6
7  /* The following two functions don't need trapb/excb instructions
8     around the mf_fpcr/mt_fpcr instructions because (a) the kernel
9     never generates arithmetic faults and (b) call_pal instructions
10    are implied trap barriers.  */
11
12 static inline unsigned long
13 rdfpcr(void)
14 {
15     unsigned long tmp, ret;
16
17 #if defined(CONFIG_ALPHA_EV6) || defined(CONFIG_ALPHA_EV67)
18     __asm__ __volatile__ (
19         "ftoit $f0,%0\n\t"
20         "mf_fpcr $f0\n\t"
21         "ftoit $f0,%1\n\t"
22         "itoft %0,$f0"
23         : "=r"(tmp), "=r"(ret));
24 #else
25     __asm__ __volatile__ (
26         "stt $f0,%0\n\t"
27         "mf_fpcr $f0\n\t"
28         "stt $f0,%1\n\t"
29         "ldt $f0,%0"
30         : "=m"(tmp), "=m"(ret));
31 #endif
32
33     return ret;
34 }
35
36 static inline void
37 wrfpcr(unsigned long val)
38 {
39     unsigned long tmp;
40
41 #if defined(CONFIG_ALPHA_EV6) || defined(CONFIG_ALPHA_EV67)
42     __asm__ __volatile__ (
43         "ftoit $f0,%0\n\t"
.../linux/arch/alpha/include/asm/fpu.h  [Ins]                    ( 1, 0) [Top/1.8k] [
Eval: START
```

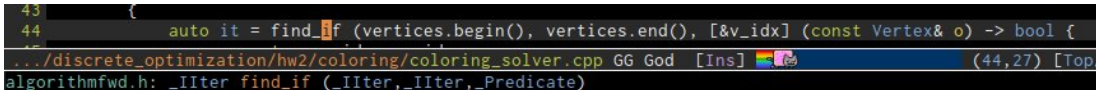
There are more, you can find it in my Projectile guide. You can also run C-c p C-h for a list of commands with C-c p prefix.

Source code information

Command: `global-semantic-idle-summary-mode`

Table of Contents

This mode, when enabled, displays function interface in the minibuffer:



It works well for C but not C++, since C++ can overload function and `semantic-idle-summary-mode` can only display one interface at a time. Since this mode is part of Semantic, it also relies on SemanticDB that is created from parsing source files.

Setup:

```
(global-semantic-idle-summary-mode 1)
```

Command: `global-semantic-stickyfunc-mode`

When enable, this mode shows the function point is currently in at the first line of the current buffer. This is useful when you have a very long function that spreads more than a screen, and you don't have to scroll up to read the function name and then scroll down to original position.

Table of Contents

```
     * Set up kernel memory allocators
464   */
465  static void __init mm_init(void)
466  {
467      /*
468       * page_cgroup requires contiguous pages,
469       * bigger than MAX_ORDER unless SPARSEMEM.
470       */
471      page_cgroup_init_flatmem();
472      mem_init();
473      kmem_cache_init();
474      percpu_init_late();
475      pgtable_init();
476      vmalloc_init();
477  }
478
479  asmlinkage __visible void __init start_kernel(void)
480  {
481      char * command_line;
482      extern const struct kernel_param __start___param[], __stop___param[];
483
484      /*
485       * Need to run as early as possible, to initialize the
486       * lockdep hash:
487       */
488      lockdep_init();
489      smp_setup_processor_id();
490      debug_objects_early_init();

492      /*
493       * Set up the the initial canary ASAP:
494       */
495      boot_init_stack_canary();
496
497      cgroup_init_early();
498
499      local_irq_disable();
500      early_boot_irqs_disabled = true;
501
502  /*
503   * Interrupts are still disabled. Do necessary setups, then
504   * enable them
505   */
506      boot_cpu_init();
507      page_address_init();
508      pr_notice("%s", linux_banner);
509      setup_arch(&command_line);
510      mm_init_owner(&init_mm, &init_task);
511      mm_init_cpumask(&init_mm);
512      setup_command_line(command_line);
513      setup_nr_cpu_ids();
514      setup_per_cpu_areas();
515      smp_prepare_boot_cpu(); /* arch-specific boot-cpu hooks */
~/linux/init/main.c  [Ins] (test)                    (479, 0) [52%/23k]
Eval: START
```

Table of Contents

Setup:

```
(add-to-list 'semantic-default-submodes 'global-semantic-stickyfunc-mode)
```

One of the problem with current `semantic-stickyfunc-mode` is that it does not display all parameters that are scattered on multiple lines. This package handles that problem: semantic-stickyfunc-enhance. Extra: stock `semantic-stickyfunc-mode` does not include assigned values to function parameters of Python. This package also fixed that problem. You can click the link to see demos.

You can install the package via MELPA and load it:

```
(require 'stickyfunc-enhance)
```

Using `ggtags` + `eldoc`

You can also use `ggtags` for displaying function interface at point in minibuffer:

```
(setq-local eldoc-documentation-function #'ggtags-eldoc-function)
```

However, it won't have syntax highlighting.

Source code documentation

Command: `man`

To read the documentation for symbol at point or a man entry of your choice, invoke `M-x man`.

Command: `helm-man-woman`

If you use Helm, you should be able to use the command `helm-man-woman`, which is bound to C-c h m if you follow my Helm guide. The full guide for the command is here.

Table of Contents

```
DEFUN ("dump-emacs", Fdump_emacs, Sdump_emacs, 2, 2, 0,
2143       Meanwhile, my_edata is not valid on Windows.  */
2144     {
2145       extern char my_edata[];
2146       memory_warnings (my_edata, malloc_warning);
2147     }
2148   #endif /* not WINDOWSNT */
2149   #endif /* not SYSTEM_MALLOC */
2150   #ifdef DOUG_LEA_MALLOC
2151     malloc_state_ptr = malloc_get_state ();
2152   #endif
2153
2154     unexec (SSDATA (filename), !NILP (symfile) ? SSDATA (symfile) : 0);
2155
2156   #ifdef DOUG_LEA_MALLOC
2157     free (malloc_state_ptr);
2158   #endif
2159
2160   #ifdef WINDOWSNT
2161     Vlibrary_cache = Qnil;
2162   #endif
2163   #ifdef HAVE_WINDOW_SYSTEM
2164     reset_image_types ();
2165   #endif
2166
2167     Vpurify_flag = tem;
2168
2169     return unbind_to (count, Qnil);
2170   }
2171
2172   #endif /* not CANNOT_DUMP */
2173   ^L
2174   #if HAVE_SETLOCALE
2175   /* Recover from setlocale (LC_ALL, "").  */
2176   void
2177   fixup_locale (void)
2178   {
2179     /* The Emacs Lisp reader needs LC_NUMERIC to be "C",
2180        so that numbers are read and printed properly for Emacs Lisp.  */
2181     setlocale (LC_NUMERIC, "C");
2182   }
2183
2184   /* Set system locale CATEGORY, with previous locale *PLOCALE, to
2185      DESIRED_LOCALE.  */
~/Downloads/emacs/src/emacs.c  [Ins]        (2157, 5) [82%/77k] [C/l] company [Git-master]  [dtrt-indent adjus
Eval: START
```

Source code editing

Folding

Emacs has a minor mode called `hs-minor-mode` that allows users to fold and hide blocks of text. Blocks are defined by regular expressions which match the start and end of a text region. For example, anything in between `{` and `}` is a block. The regular expressions are defined in

Table of Contents

`hs-special-modes-alist`.

Setup for C/C++:

```
(add-hook 'c-mode-common-hook   'hs-minor-mode)
```

Default key bindings:

Key	Binding
C-c @ C-c	Command: `hs-toggle-hiding` Toggle hiding/showing of a block
C-c @ C-h	Command: `hs-hide-block` Select current block at point and hide it
C-c @ C-l	Command: `hs-hide-level` Hide all block with indentation levels below this block
C-c @ C-s	Command: `hs-show-block` Select current block at point and show it.
C-c @ C-M-h	Command: `hs-hide-all` Hide all top level blocks, displaying only first and last lines.
C-c @ C-M-s	Command: `hs-show-all` Show everything

Demo:

- Command: `hs-toggle-hiding`

Table of Contents

```
 * For ex. kdump situaiton where previous kernel has crashed, BIOS has been
 * skipped and devices will be in unknown state.
 */
unsigned int reset_devices;
EXPORT_SYMBOL(reset_devices);

static int __init set_reset_devices(char *str)
{
    reset_devices = 1;
    return 1;
}

__setup("reset_devices", set_reset_devices);

static const char * argv_init[MAX_INIT_ARGS+2] = { "init", NULL, };
const char * envp_init[MAX_INIT_ENVS+2] = { "HOME=/", "TERM=linux", NULL, };
static const char *panic_later, *panic_param;

extern const struct obs_kernel_param __setup_start[], __setup_end[];

static int __init obsolete_checksetup(char *line)
{
    const struct obs_kernel_param *p;
    int had_early_param = 0;

    p = __setup_start;
    do {
        int n = strlen(p->str);
        if (parameqn(line, p->str, n)) {
            if (p->early) {
                /* Already done in parse_early_param?
                 * (Needs exact match on param part).
                 * Keep iterating, as we can have early
                 * params and __setups of same names 8( */
                if (line[n] == '\0' || line[n] == '=')
                    had_early_param = 1;
            } else if (!p->setup_func) {
                pr_warn("Parameter %s is obsolete, ignored\n",
                    p->str);
                return 1;
            } else if (p->setup_func(line + n))
                return 1;
        }
        p++;
    } while (p < __setup_end);

    return had_early_param;
}

/*
 * This should be approx 2 Bo*oMips to start (note initial shift), and will
 * still work even if initially too large, it will just take slightly longer
 */
~/linux/init/main.c  [Ins] (test)                    (168, 1) [19%/23k]
Declaration Modifier: const <type> <name> ...
```

Table of Contents

Narrowing

Narrowing means making only a text portion in current buffer visible. Narrowing is useful when you want to perform text editing on a small part of the buffer without affecting the others. For example, you want to delete all `printf` statements in current functions, using `flush-lines` command. But if you do so, you will also delete `printf` outside the current function, which is undesirable. By narrowing, you can safely remove all those `printf` and be certain that nothing else is changed accidentally.

Default key bindings:

Key	Binding
C-x n d	Command: `narrow-to-defun` Narrow buffer to current function at point
C-x n r C-x n n	Command: `narrow-to-region` Narrow buffer to active region
C-x n w	Command: `widen` Widen buffer

Identation

Setup default C style

Emacs offers some popular C coding styles. Select the one suitable for you:

```
;; Available C style:
;; “gnu”: The default style for GNU projects
;; “k&r”: What Kernighan and Ritchie, the authors of C used in their book
;; “bsd”: What BSD developers use, aka “Allman style” after Eric Allman.
;; “whitesmith”: Popularized by the examples that came with Whitesmiths C, an early commercial C compiler.
;; “stroustrup”: What Stroustrup, the author of C++ used in his book
;; “ellemtel”: Popular C++ coding standards as defined by “Programming in C++, Rules and Recommendations,” Erik Nyquist and Mats Henricson, Ellemtel
;; “linux”: What the Linux developers use for kernel development
;; “python”: What Python developers use for extension modules
;; “java”: The default style for java-mode (see below)
;; “user”: When you want to define your own style
(setq
 c-default-style "linux" ;; set style to "linux"
 )
```

Table of Contents

Setup indentation

By default, Emacs won't indent when press RET because the command bound to RET is newline. You can enable automatic indentation by binding RET to `newline-and-indent`.

```
(global-set-key (kbd "RET") 'newline-and-indent)  ; automatically indent when press RET
```

When working with source code, we must pay attention to trailng whitespace. It is always useful to view whitespace in current buffer before committing your code.

```
;; activate whitespace-mode to view all whitespace characters
(global-set-key (kbd "C-c w") 'whitespace-mode)

;; show unncessary whitespace that can mess up your diff
(add-hook 'prog-mode-hook (lambda () (interactive) (setq show-trailing-whitespace 1)))

;; use space to indent by default
(setq-default indent-tabs-mode nil)

;; set appearance of a tab that is represented by 4 spaces
(setq-default tab-width 4)
```

To clean up trailing whitespace, you can also run `whitespace-cleanup` command.

To convert between TAB and space, you also have two commands: `tabify` to turn an active region to use TAB for indentation, and `untabify` to turn an active region to use space for indentation.

Package: `clean-aindent-mode`

When you press RET to create a newline and got indented by `eletric-indent-mode`, you have appropriate whitespace for indenting. But, if you leave the line blank and move to the next line, the whitespace becomes useless. This package helps clean up unused whitespace.

View this Emacswiki page for more details.

```
;; Package: clean-aindent-mode
(require 'clean-aindent-mode)
(add-hook 'prog-mode-hook 'clean-aindent-mode)
```

Package: `dtrt-indent`

Table of Contents

A minor mode that guesses the indentation offset originally used for creating source code files and transparently adjusts the corresponding settings in Emacs, making it more convenient to edit foreign files.

This package is really useful when you have to work on many different projects (for example, your project consists of many git submodules) that use different indentation format. One project might use TAB for indentation; another project might use space for indentation. Having to manually switch between TAB and space is tiresome. We should let Emacs take care of it automatically. This package does exactly that. However, if a project mixes TAB and space, then the package will have a hard time to detect. But then again, if you work on such project, it doesn't matter anyway.

Setup:

```
;; Package: dtrt-indent
(require 'dtrt-indent)
(dtrt-indent-mode 1)
```

If you use Semantic for code completion, you may one to turn off `dtrt-indent` messages since Semantic may visit many files for parsing, and each file entered a message from `dtrt-indent` is printed in echo area and might become a constant annoyance.

```
(setq dtrt-indent-verbosity 0)
```

Package: `ws-butler`

`ws-butler` helps managing whitespace on every line of code written or edited, in an unobtrusive, help you write clean code without noisy whitespace effortlessly. That is:

- Only lines touched get trimmed. If the white space at end of buffer is changed, then blank lines at the end of buffer are truncated respecting require-final-newline.
- Trimming only happens when saving.

With `clean-aindent-mode` and `dtrt-indent` and `ws-butler`, you could totally forget about whitespace. If you are careful, turn on `whitespace-mode` (bound to `C-c w` above) and check for whitespace before committing.

Setup:

```
  ;; Package: ws-butler
(require 'ws-butler)
(add-hook 'c-mode-common-hook 'ws-butler-mode)
```

Code template using `yasnippet`

YASnippet is a template system for Emacs. It allows you to type an abbreviation and automatically expand it into function templates. Bundled

Table of Contents

language templates include: C, C++, C#, Perl, Python, Ruby, SQL, LaTeX, HTML, CSS and more. The snippet syntax is inspired from TextMate's syntax, you can even import most TextMate templates to YASnippet.

Setup:

```
;; Package: yasnippet
(require 'yasnippet)
(yas-global-mode 1)
```

Usage:

In major modes where yasnippet has snippets available, typing a certain keyword and TAB insert a predefined snippet. For example, in a C buffer, if you type `for` and TAB, it expands to:

```
for (i = 0; i < N; i++) {
    ...point will be here....
}
```

You can view supported snippets here.

Package: `smartparens`

`smartparens` is a minor mode that provides many features for manipulating pairs. Pair can be simple as parentheses or brackets, or can be programming tokens such as `if` ... `fi` or `if` ... `end` in many languages. The most basic and essential feature is automatic closing of a pair when user inserts an opening one.

```
  ;; Package: smartparens
(require 'smartparens-config)
(show-smartparens-global-mode +1)
(smartparens-global-mode 1)

;; when you press RET, the curly braces automatically
;; add another newline
(sp-with-modes '(c-mode c++-mode)
  (sp-local-pair "{" nil :post-handlers '(("||\n[i]" "RET")))
  (sp-local-pair "/*" "*/" :post-handlers '((" | " "SPC")
                                            ("* ||\n[i]" "RET"))))
```

For complete documentation, please refer to Smartparens manual.

Table of Contents

Compilation Support

Compilation mode turns each error message in the buffer into a hyperlink. You can click on each error, or execute a key binding like RET to jump to the location of that error.

The following key bindings are available:

Key	Description
C-o	Display matched location, but do not switch point to matched buffer
M-n	Move to next error message, but do not visit error location
M-p	Move to next previous message, but do not visit error location
M-g n	Move to next error message, visit error location
M-g p	Move to previous error message, visit error location
RET	Visit location of error at poiint
M-{	Move point to the next error message or match occurring in a different file
M-}	Move point to the previous error message or match occurring in a different file
q	Quit `*compilation*` buffer

I usually execute the same compilation command many times. It's more convenient if Emacs doesn't ask us to confirm every time we re-execute a command:

```
(global-set-key (kbd "<f5>") (lambda ()
                              (interactive)
                              (setq-local compilation-read-command nil)
                              (call-interactively 'compile)))
```

If you want to enter a new command, add prefix argument before pressing `<f5>`.

Demo:

Table of Contents

```
/home/tuhdo/Dropbox/Projects/s3183594_s3245863_Le Duc Tuan_Do Hoang Tu_COSC2131_A2/Terminal:
.
..
.svn
basketball.txt
chemical_substance.txt
#Crossword.cpp#
Crossword.cpp
Crossword.h
CrosswordLib.h
Crossword.o
Letter.cpp
Letter.h
Letter.o
main.cpp
main.o
Makefile
Position.cpp
Position.h
Position.o
sample_data1.txt
sample_data2.txt
sample_data3.txt
sample_data4.txt
science.txt
solarsystem_crossword_puzzle.txt
#Word.cpp#
Word.cpp
Word.h
Word.o

.../s3183594_s3245863_Le Duc Tuan_Do Hoang Tu_COSC2131_A2/Terminal/Terminal  [Ins,Mod,RO] (test)  ( 5, 2
Eval: START
```

Table of Contents

As you can see, output in `*compilation*` buffer looks much better than in plain terminal with colors and you can also interact with it.

You can also run other shell commands in here, not just compilation commands. A really cool thing with `compilation-mode` is that it is asynchronous. That is, if your source files contain errors, you don't have to wait for the whole compilation process to finish. You can examine the errors right away in the `*compilation*` buffer, while the compilation process keeps going.

When you compile with `compilation-mode`, your shell output is processed and highlighted with beautiful colors to easy your reading:

```
checking for cp... cp
checking for ln... ln
checking for tar... tar
checking for rpmbuild... rpmbuild
checking for sed... sed
checking for find... find
checking for xargs... xargs
checking for dirname... dirname
checking for grep that handles long lines and -e... /usr/bin/grep
checking for egrep... /usr/bin/grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking zlib.h usability... yes
checking zlib.h presence... yes
checking for zlib.h... yes
checking for inflateInit_ in -lz... yes
checking lzma.h usability... no
checking lzma.h presence... no
checking for lzma.h... no
checking xenctrl.h usability... no
checking xenctrl.h presence... no
checking for xenctrl.h... no
configure: creating ./config.status
config.status: creating Makefile
config.status: creating include/config.h
+ make
gcc -fmessage-length=0 -O2 -Wall -D_FORTIFY_SOURCE=2 -fstack-protector -funwind-tables -fasynchronou
nclude -I./util_lib/include -Iinclude/ -I./kexec/arch/x86_64/libfdt -I./kexec/arch/x86_64/include  -
kexec/kexec.c: In function 'main':
kexec/kexec.c:1229: warning: implicit declaration of function 'xen_balloon_up'
 /local/xtuudoo/enux/*compilation*  [Ins,Mod,RO]  (1473, 0) [26%/304k] [Compilation] company
```

TIP: It is useful to open a Dired buffer of the project root and store a window configuration that contains it into a register, so you can always jump back to project root any where, any time and run compilation command. If you do not use know how to use registers to store window

configurations, read it here.

Table of Contents

Debugging

With GDB Many Windows

Emacs has built-in frontend support for GDB that provides IDE-like interface. It has a layout like this:

Table of Contents

```
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bufbomb...done.
(gdb) b test
Breakpoint 1 at 0x400ed0: file bufbomb.c, line 103.
(gdb) r -u 2866
Starting program: /home/tuhdo/course-materials/lab3/bufbomb -u 2866
(gdb)
```

```
Locals Registers
unsigned long long          val             <optimized out>
volatile unsigned long long local           140737351856328
char *                      variable_length <optimized out>
```

~/course-materials/lab3/*gud-bufbomb* [Ins,Mod] (test)

```
     void fizz(int arg1, char arg2, long arg3, char* arg4, short arg5, short arg6, unsign
 79          printf("Misfire: You called fizz(0x%llx)\n", val);
 80      }
 81      exit(0);
 82  }
 83  /* $end fizz-c */

 85  /* $begin bang-c */
 86  unsigned long long global_value = 0;
 87
 88  void bang(unsigned long long val)
 89  {
 90      entry_check(2);  /* Make sure entered this function properly */
 91      if (global_value == cookie) {
 92          printf("Bang!: You set global_value to 0x%llx\n", global_value);
 93          validate(2);
 94      } else {
 95          printf("Misfire: global_value = 0x%llx\n", global_value);
 96      }
 97      exit(0);
 98  }
 99  /* $end bang-c */
100
101  /* $begin boom-c */
102  void test()
103  {
```

~/course-materials/lab3/*locals of bufbomb* [Ins,Mod,RO]

```
Username: 2866
96
95
64
7e
ac
a
ce
8c
91
5f
1d
55
48
4c
Cookie: 0x4c48551d5f918cce
```

~/course-materials/lab3/bufbomb.c [Ins] (test) (103, 0)

```
0 in test of bufbomb.c:103
1 in launch of bufbomb.c:343
2 in main of bufbomb.c:454
```

~/course-materials/lab3/*input/output of bufbomb* [Ins,Mod

```
Breakpoints Threads
Num Type       Disp Enb Addr               Hits What
1   breakpoint keep y   0x0000000000400ed0 1    in test of
```

~/course-materials/lab3/*stack frames of bufbomb* [Ins,Mod,RO] (test)

~/course-materials/lab3/*breakpoints of bufbomb* [Ins,Mod,

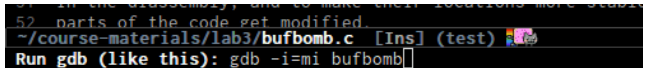
```
Eval: START
```

Table of Contents

Stock Emacs doesn't enable this layout by default. You have to tell Emacs to always use `gdb-many-windows`:

```
(setq
 ;; use gdb-many-windows by default
 gdb-many-windows t

 ;; Non-nil means display source file containing the main routine at startup
 gdb-show-main t
 )
```

Now, find a binary built for debugging, and start GDB by `M-x gdb`. Emacs prompts asking you how to run `gdb`. By default, the prompt looks like this:



To use `gdb-many-windows`, you must always supply the -i=mi argument to gdb, otherwise `gdb-many-windows` won't work. By default, as you've seen the layout of `gdb-many-windows` above, you have the following buffers visible on your screen (ignore the first row):

1. GUD interaction buffer	2. Locals/Registers buffer
3. Primary Source buffer	4. I/O buffer for debugging program
5. Stack buffer	6. Breakpoints/Threads buffer

Each cell coressponds to the following commands:

- `gdb-display-gdb-buffer` (Cell 1): This is where you enter `gdb` commands, the same as in terminal. When you kill this buffer, other GDb buffers are also killed and debugging session is terminated.
- `gdb-display-locals-buffer` (Cell 2): display local variables and its values in current stack frame. Please refer to GNU Manual - Other GDB Buffers for usage of the buffer.
- `gdb-display-registers-buffer` (Cell 2): registers values are displayed here. Please refer to GNU Manual - Other GDB Buffers for usage of the buffer.
- Your source buffer (Cell 3): Your source code for stepping through out the debugging session. Please refer to GNU Manual - Source Buffers for usage of the buffer.
- `gdb-display-io-buffer` (Cell 4): This is where your program displays output and accepts input. In stock GDB (Command Line Interface), you enter input whenever a program asks for one under GDB prompt. Using `gdb-many-windows`, you must enter program input here.

Table of Contents

- `gdb-display-stack-buffer` (Cell 5): Display function call stack. Please refer to GNU Manual - Stack buffer for usage.
- `gdb-display-breakpoints-buffer` (Cell 6): Display a list of breakpoints. Please refere to GNU Manual - Breakpoints Buffer for usage of the buffer.
- `gdb-display-threads-buffer` (Cell 6): Display running threads in your programs. Please refer to GNU Manual - Threads Buffer and Multithreaded Debugging for usage of the buffer.

There are two useful commands that are not visible in `gdb-many-windows`:

- `gdb-display-disassembly-buffer`: displays the current frame as assembly code.
- `gdb-display-memory-buffer`: displays a portion of program memory.

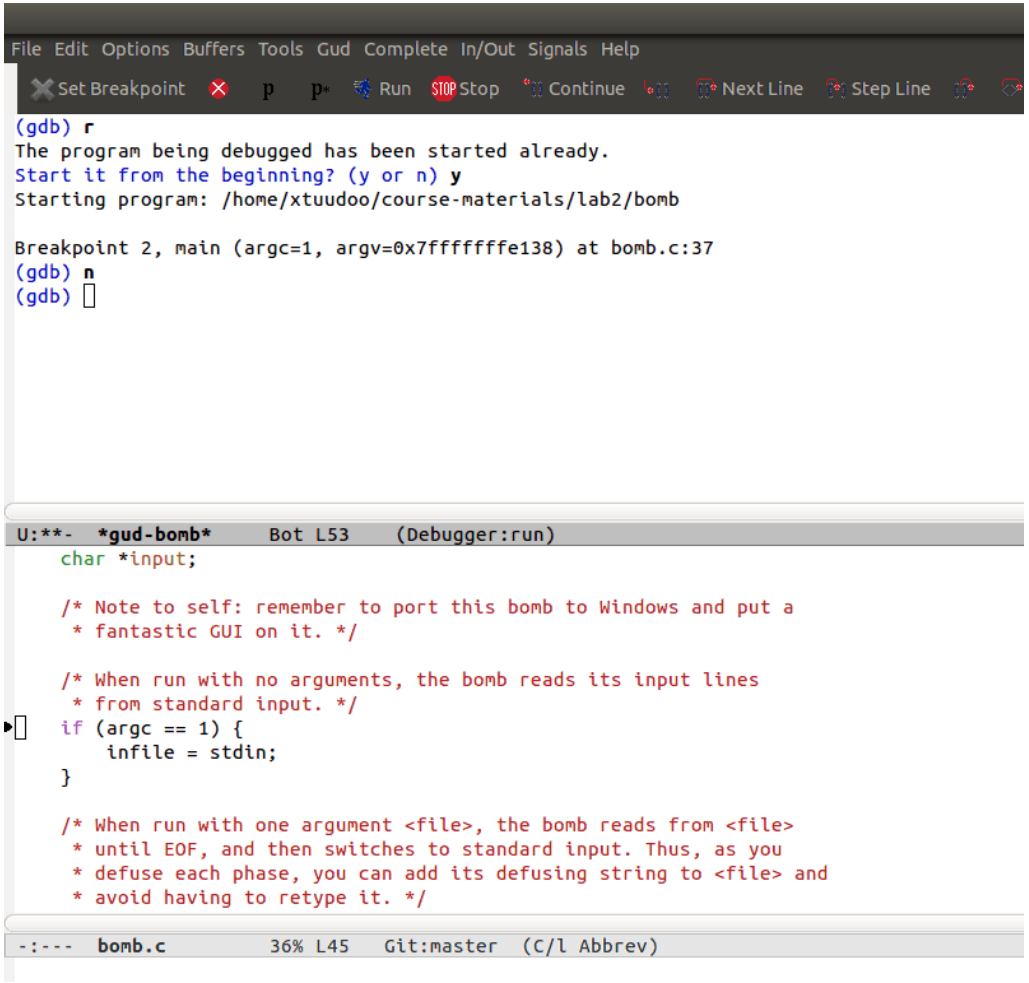
There are another variants of the above buffers, with `gdb-frame` prefix instead of `gdb-display`. When you run commands with `gdb-frame` prefix, the buffers are opened in a new frame instead of occupying an existing window. This is useful when you have 2 monitors, one is for the standard `gdb-many-windows` layout and another one is for displaying disassembly buffer and memory buffer in a separate frame.

With Grand Unified Debugger - GUD

The Grand Unified Debugger, or GUD for short, is an Emacs major mode for debugging. It works on top of command line debuggers. GUD handles interaction with gdb, dbx, xdb, sdb, perldb, jdb, and pdb, by default. Emacs provides unified key bindings for those debuggers. We only concern about GDB in this guide.

GUD works by sending text commands to a debugger subprocess, and record its output. As you debug the program, Emacs displays relevant source files with an arrow in the left fringe indicating the current execution line. GUD is simple and stable. When starting GUD, you only have a GDB command buffer and your source file. If you like simplicity, you can use GUD. However, you can still use other `gdb-display-` or `gdb-frame-` variants listed in previous section when needed.

Table of Contents



As you can see, the default Emacs interface also includes a menu for regular debugging operations, such as Stop, Run, Continue, Next Line, Up/Down Stack...

Refer to a list of commands supported by GUD at GNU Emacs Manual - Commands of GUD. Note that these key bindings not only work with GDB, but any debuggers supported by GUD, since it's a Grand Unified Debugger.

There's an Emacs packages that offers support for more debuggers: emacs-dbgr. Here is the list of supported debuggers. If you have to work with

many languages, use GUD.

Table of Contents

50 Comments http://tuhdo.github.io/emacs-tutor.html 1 Login

Recommend 1 Share Sort by Best

Join the discussion…

alastair • a year ago

This is really great. I'm glad you included my company-c-headers mode.

I tried to write something like this myself. The initial attempt is here, might have something of use for you: https://github.com/randomphras...

1 • Reply • Share ›

tuhdo Mod → alastair • a year ago

Thanks. I'll take a look at your guide.

company-c-headers is useful and was in my toolchain for a long time ago.

Cheer.

• Reply • Share ›

Oleg • a year ago

Please, use thumbnails. Page very slow. And use another font, sans-serif one.

1 • Reply • Share ›

tuhdo Mod → Oleg • a year ago

Hi,

You can clone this entire website (including other guides) for reading offline here: https://github.com/tuhdo/tuhdo...

As for font, I will look into it.

EDIT: or you can read in Github that it resizes to fit a page: https://github.com/tuhdo/tuhdo... . However, you won't have a table of contents for navigation though.

• Reply • Share ›

Meir Goldenberg • 4 months ago

I was not able to make the completion of C++ headers to work. When I set 'company-c-headers-path-system using add-to-list, Emacs complains about a void variable. So, I tried to use both the setq syntax and by M-x customize-group which resulted in this: '(company-c-headers-path-system (quote ("/usr/include/c++/4.8/" "/usr/include/c++/4.8.2/")))). In both cases no complaints, but still no completion... The directories do exist and contain the headers... Using Emacs 24.5 under Ubuntu 14.04. company-c-headers is installed. Thank you so much for the tutorial!

• Reply • Share ›

Meir Goldenberg → Meir Goldenberg • 4 months ago

Oops. I was testing it on a .c file! It looks like the package does not look at non-.h files when #include is used in a .c file!

• Reply • Share ›

anacy • 8 months ago

Can I ask you what color theme you are using on the first 2 pictures?

• Reply • Share ›

Table of Contents

Table of Contents