Advanced C Programming

Editors, Debug Macros

Sebastian Hack
hack@cs.uni-sb.de

Christoph Weidenbach
weidenbach@mpi-inf.mpg.de

02.12.2008

Contents

XEmacs

XEmacs: `http://www.xemacs.org/`

- evolved from GNU emacs
- differences between emacs/xemacs mainly only show up for advanced users
- "there are currently irreconcilable differences in the views about technical, programming, design and organizational matters between Richard Stallman (RMS) and the XEmacs development team which provide little hope for a merge to take place in the short-term future"
- self-documenting (<CTRL-h>), customizable, extensible real-time editor mainly written in LISP

XEmacs

Basic Concepts

- Buffer: a region of memory holding characters, basic editing unit
- File: a region of disk space holding characters
- Window: rectangular region in which a buffer is displayed

XEmacs

Identation

- <TAB> does identation at current cursor position
- <CTRL-META-q> idents a balanced brace
- <CTRL-META-\ > idents a region that can be marked by the mouse are a <CTRL-SPACE> (start region) moving sequence or <CTRL-META-h> for the current function
- the default indentation can be adjusted

XEmacs

Tagging

- after building a TAG database using the command "etags $*$.[ch]"
- <META-.> jumps to the current function's definition
- <CTRL-c s c> finds/jumps to callers of the current function

XEmacs

Debugging

- <META-x gdb> activates gdb inside xemacs
- <CTRL-x SPACE> creates a breakpoint in any source file
- then sveral gdb commands are available as keystrokes: <META-i> executes one instruction, <META-c> continues the program, ...

XEmacs

Navigation: Forward (analogous backward, deleting, marking)

- <CTRL-f> go forward one letter
- <META-f> go forward one word
- <CTRL-e> go forward to end of line
- <META-x c-end-of-defun> go forward to end of definition (key binding)
- <CTRL-v> go forward one page
- <META->> go forward end of buffer

XEmacs

Diff Support for Files and Buffers

Commenting

- <CTRL-c CTRL-c> comment region
- <CTRL-u CTRL-c CTRL-c> uncomment region

Vertical Editing

- <CTRL-x r k> kill rectangle
- <CTRL-x r y> yank rectangle
- <CTRL-x r t> string add rectangle

XEmacs

Basic Keyboard Macro Definitions

- <CTRL-x (> <define your macro> <CTRL-x)>
- <CTRL-x e> repeat recorded macro
- <CTRL-x CTRL-k b> define a key sequence for the macro
- <M-x name-last-kbd-macro> name last macro
- <M-x insert-kbd-macro> insert LISP code for macro identified by name into buffer

Vi(m): Basic concepts

- "Visual" extension of line editor `ex`
  ☞ Two editors in one
- Mode-based editing: Insert mode, Normal mode
- Normal mode is for navigating and searching
- Insert mode for changing
- Editing is "transactional"
  Can be easily undone, replayed, recorded, etc.
- Advantage: "Transaction granularity" is controllable by user
- Editing commands use normal keys of the keyboard
- To enter edit mode:
  - `i` (insert) Start inserting at cursor
  - `I` Insert at start of line
  - `R` OverwRite
  - `A` (append) Goto end of line and insert
  - `s` (substitute) Delete current character and insert
  - `C` Delete from cursor to rest of line and insert
- Press <ESC> to exit insert mode

Vi(m): Movement

- Movement `h, j, k, l` is left, down, up, right
- Handy: Move on text-element basis

```
         Hello, this is just some text.
|        |                 || | | X| ||| |
0        ^                 ||ge b  e w|| g_      $
                         Fj|           |fx
                           Tj          tx
```

- Many commands can be preceded by a count that specifies how often the command is executed
- Movement can be combined with editing
  - `dw` Delete until start of next word
  - `3dw` Do so 3 times
  - `ct{` Delete everything up to the next open brace (excluded) and go to insert
  - `yf:` Copy everything up to the next `:` into the clipboard

Marks and Registers

Marks

- Use marks to navigate
- 26 inner-file marks (`a-z`)
- 26 across-file marks (`A-Z`)
- `mx` to set mark `x` to current position
- `` `x `` to jump to position in mark
- `` `. `` jump to place of last change (very useful!)
- `` `x `` is a movement command; can combine it with editing:
  ☞ e.g. `d`a Delete everything up to mark a
- Visual mode selections define marks `'<` and `'>`

Registers

- Containers for Text
- Yank and paste to/from them
- `"xy`[motion] yank to register `x`
- `"xp`[motion] paste from register `x`
- Don't need to specify the default register

Vim for Programmers

- =[movement] Indents the specified text
- A simple `:make` invokes `make`
  - Parses error list afterwards
  - Can go through one by one
- Simple Navigation
  - `[{` and `]}` to jump to enclosing opening (closing) brace
  - `{` and `}` to go to next paragraph
  - `gd` to goto definition of local variable
  - `%` to jump to matching part (of brace)
    - works with all kinds of parens, braces, ...
    - `#if #endif`
    - Install `matchit.vim` for more
- Tags
  - Use `ctags -R *` to generate a `tags` file
  - Put a `tags` rule in your Makefile
  - Vim loads it automatically
  - Use `C-]` to go to definition of symbol
  - Use `C-t` to go back

Macros

- You can define macros `a-z`
- Start recording macro `x` with `qa`
- Compose a sequence of events
- Stop recording by hitting `q` again
- Execute the macro with `@x`
- And re-execute the last macro with `@@`
- Of course you can use that with a count: `25@@`

Nice Helpers

- `.` repeats last command
- `rX` replaces the current character `X` without going to insert mode
- `~` changes the case of current char or selection
- `J` join two lines
- `o` (`O`) start a new line below (above) the current one
- Repeating some commands apply them on a line: `yy`, `cc`, `dd`, `==`
- Search is also movement: `my_text`
  ☞ `d/my_text` deletes everything up to the next hit on `my_text`
- `x` deletes character: `xp` exchanges two of them; logical!
- `u` undo is your friend!
- `v` and `V` for visual selection mode (select text with the cursor)
- `C-v` for rectangular visual selection mode
- `C-p` autocompletes based on text above
- Vim 7 also has some context-sensitive completion mapped on `C-x C-o`
  remap to `C-Space`: `inoremap <Nul> <C-x><C-o>`

Contents

Debug Macros

- The best debugger is `printf` :-)

```
    z = x + y;
    printf("z␣=␣%d\n", z);
```

- However, we do want to
  - have only certain printfs of parts that interest us
  - not have all printfs in the release code ☞ slow

```
    z = x + y;
#ifdef DEBUG_MODULE_A
    printf("z␣=␣%d\n", z);
#endif /* DEBUG_MODULE_A */
```

- But, we do not want write these `ifdefs` all over the place
- Something more like

```
    z = x + y;
    debug_module_a("z␣=␣%d\n", z);
```

or

```
    debug(module_a, "z␣=␣%d\n", z);
```

Debug Macros

- Because we want no overhead in the release build, debug must be a macro
- Put it in uppercase

```
    z = x + y;
    DBG("z␣=␣%d\n", x + z);
```

- Not very portable because requires C99 vararg macros
- A better way:

```
    z = x + y;
    DBG(("z␣=␣%d\n", x + z));
```

- How does the declaration look like?

```
#ifdef PRGDEBUG
#define DBG(x) printf x
#else
#define DBG(x) /* nothing */
#endif
```

- Only way to write downward-compatible vararg macros?

C90 vararg Macros

- There is another way how you can write

```
DBG("z␣=␣%d\n", z);
```

- Define DBG as follows

```
#ifdef PRGDEBUG
#define DBG printf
#else
#define DBG 1 ? 0 :
#endif
```

- Disadvantage:
  rely on the optimizer to remove code in non-debug case

- Or something even nastier

```
#define DBG(x) printf(x)
#define ARG(x) , (x)

DBG("z␣=␣%d" ARG(z));
```

Debug Macros
Adding Modules

- Add different debug "sources"

```
DBG((MOD_PARSER, "z␣=␣%d\n", z));
```

- can no longer use `printf` due to new argument

```
#ifdef PRGDEBUG
#define DBG(x) dbg_printer x
#else
#define DBG(x) /* nothing */
#endif
void dbg_printer(int module, const char *fmt, ...);
```

- How do we get file name and line number?

```
#define DBG(x) dbg_set_pos(__FILE__, __LINE__), \
               dbg_printer x

static const char *file_name;
static int line_num;

void dbg_printer(int module, const char *fmt, ...) { ... }
void dbg_set_pos(const char *file_name, int line_num) { ... }
```

- Not thread safe! ☞ put `static` variables in TLS

Debug Macros
Elegantly adding Modules

- How to add new modules elegantly
- Add a file `debug_modules.def`

```
ADD_MOD(0, PARSER)
ADD_MOD(1, SOLVER)
ADD_MOD(2, PRINTER)
```

- "Generate" an `enum` with debug modules: `debug.h`

```
...
#define ADD_MOD(num, id) MOD_ ## id = 1 << num,
enum _debug_modules_t {
#include "debug_modules.def"
};
#undef ADD_MOD
...
```

- Preprocessor yields

```
enum _debug_modules_t {
    MOD_PARSER = 1 << 0,
    MOD_SOLVER = 1 << 1,
    MOD_PRINTER = 1 << 2,
};
```

Checking

About Debug Code

- find bugs as early as possible
- any debug code is READONLY!
- use debug code for function argument checking, main invariants

Debug Code Invocation

```
#ifdef SATCHECK
if (<read only condition>) {
   <Start Debug Report Macro Call>
   <Debug Report Message Macro Call>+
   <End Debug Report Macro Call>
}
#endif
```

Checking Macros

Start Debug Report Macro

- flushes all output
- prints a generic error message to error out (tbd.) including file and line number

Debug Report Message Macro

- accepts an arbitrary printf format string with arguments
- prints the message to error out

End Debug Report Macro

- flushes all output
- dumps a core iff `SATDEBUG` is set
- exits with failure