Notes 2.0: A programmer's toolbox

COMP9021 Principles of Programming

School of Computer Science and Engineering
The University of New South Wales

2013 session 1

An option, not a must

For the labs, and for your own machine if you run Linux, I provide you with a customised version of Emacs that gives you all the benefits of an IDE when working on small programs, but is very lean, with no frills; you should feel it allows you to learn programming in an environment that is powerful but pleasant and easy to use. Mac users should use the Aquamacs, version 1.9.1 (v. 2.4 has serious problems) implementation of Emacs.

- Students who have some programming experience and are used to particular tools (editor, debugger, etc.) might want to give it a try, but are likely to prefer to stick to their familiar working environment.
- Students who have no programming experience might want to give a try to other working environments. One option is Geany. Windows users might also want and experiment with Quincy.
- So whether you are new to programming or not, you *do not have to* use the provided customised version of Emacs; you are absolutely free to use any tool you like...

The .emacs.el file

The customisation is achieved thanks to a number of files.

One of the customisation files is emacs.el

- It should reside in your home directory under the name `.emacs.el` (note the leading dot). So in the directory where you saved `emacs.el` execute

```
cp emacs.el ~/.emacs.el
```

- You might have to modify `.emacs.el` later in this course, in case you opt for a programming style different to the one I use.
- If you already have a `.emacs.el` file in your home directory, you might prefer not to replace it by the file you are provided with, but append the contents of that file to your `.emacs.el` file.

The other customisation files (1)

The other customisation files are executable perl scripts and shell scripts. It is convenient to store them in a directory such as `~/COMP9021/scripts` (the subdirectory `scripts` of the subdirectory `COMP9021` of your home directory, that first has to be created using the `mkdir` (make directory) command in case it does not exist.

Whatever that directory is, it is convenient to store it in your path. With `~/COMP9021/scripts` as example, and assuming that you have already edited your .profile file as explained in the first set of notes, you would edit `.profile` again and add `$HOME/COMP9021/scripts:` to the end of the line that starts with export `PATH=` (and run `. .profile` if you want the change to take effect without having to log out first).

There are three other customisation files to save in that directory:

`_mctemplate` `_getfilenames` `_mmakefile`

The other customisation files (2)

You are unlikely to want and modify `_mmakefile` or `_getfilenames`, but you might like to modify `_mctemplate`: just change the part between quotes following `template=` so that the template suits your needs (for instance, to insert your name in the top comment).

The `_mctemplate` shell script will be called from customised Emacs in contexts where you will be given the choice between two possible templates, depending on whether you plan to write your program in a single file or in more than one file. At the beginning of the course, you are more likely to write programs in a single file and prefer the template specifically designed for that kind of program, though the other template can be used in any case, whether your program actually spans many files or not. The difference between both templates is that a line that reads

```
* Other source files, if any, one per line, ...
```

is inserted in one kind of template, and not in the other.

Using customised Emacs (1)

The `.emacs.el` file defined two elisp commands and binds them to `Control C` followed by `o` and `Control C` followed by `p`.

You can change these key bindings by editing in `.emacs.el` the two lines

```
(global-set-key "\C-c o" 'c-open-and-prepare)
(global-set-key "\C-c p" 'c-prepare)
```

`\C-c o` lets the minibuffer prompt you to enter the name of a C program; it has to end in `.c` , but the final `.c` does not have to be typed in.

- If that file exists then it is loaded into a window of the frame where the command has been entered. Then Emacs behaves as if you had typed `\C-c p`, described next.
- If that file does not exist then you are prompted to choose one of the two templates that `_mctemplate` can provide, that will then be loaded into a window.

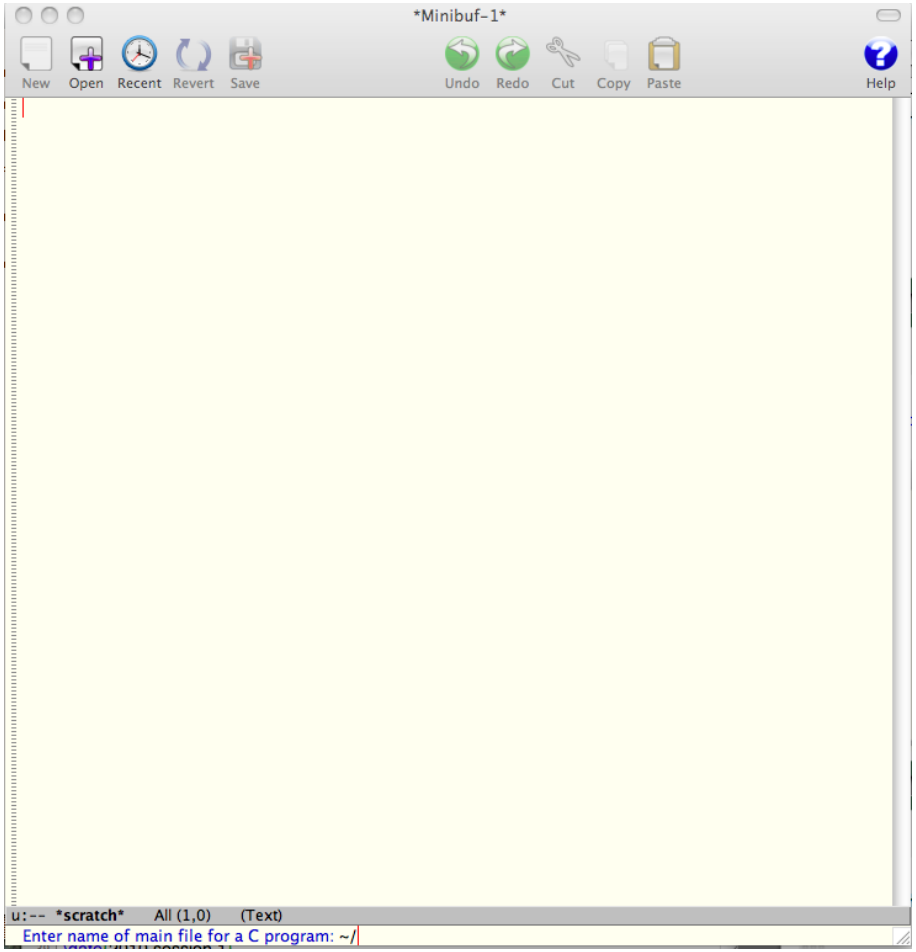
Using customised Emacs (2)

`\C-c p` lets Emacs save the file you are currently editing and compile it.

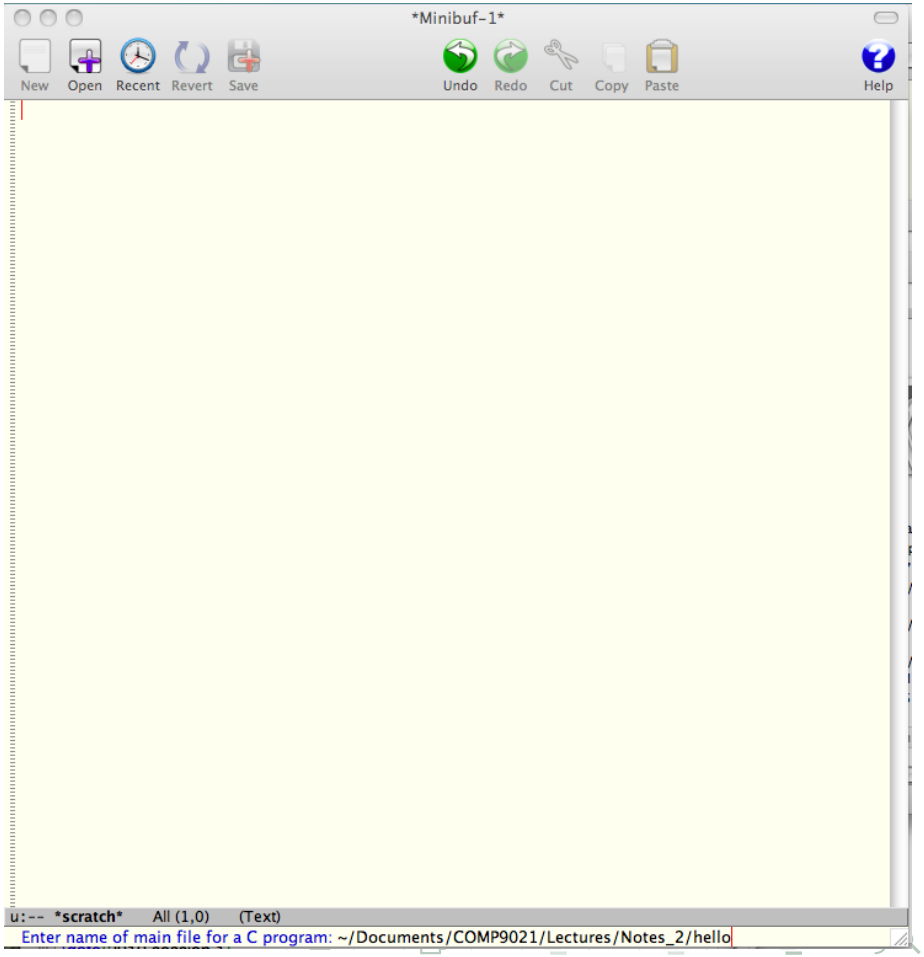
- If the compilation is unsuccessful then error messages are displayed in the `compilation` window.
- If the compilation is successful then the debugger is launched and you can run the program by typing `r` in the `gud-a.out` window.

Using customised Emacs (3)

Typing `\C-c o`

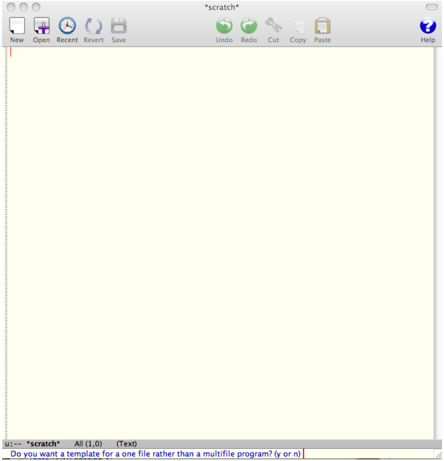


Entering the file name

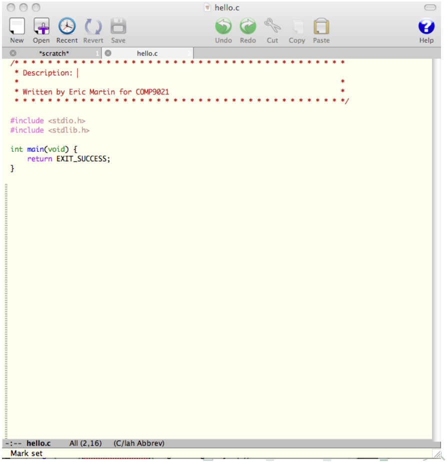


Using customised Emacs (4)

Having to choose a template

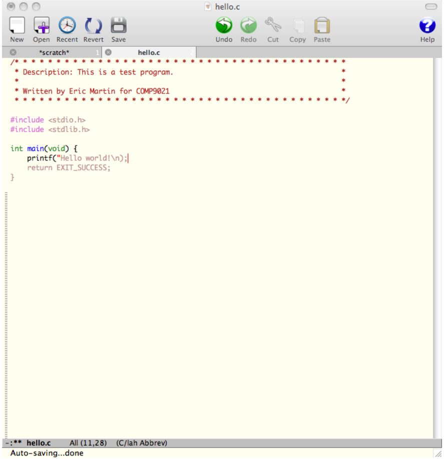


Choosing one by typing `y`

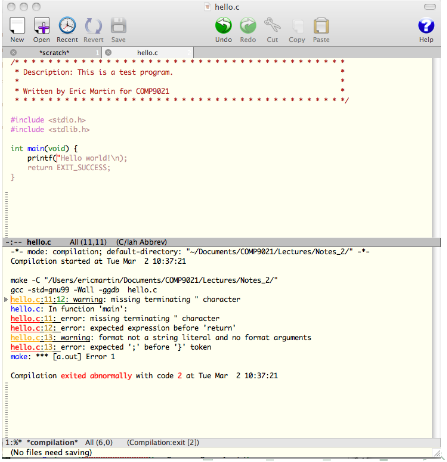


Using customised Emacs (5)

Editing the program

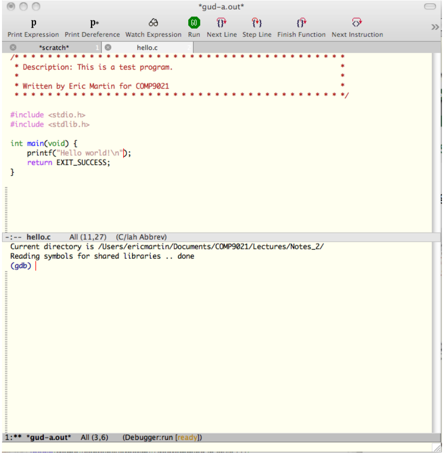


Typing `\C-c p`

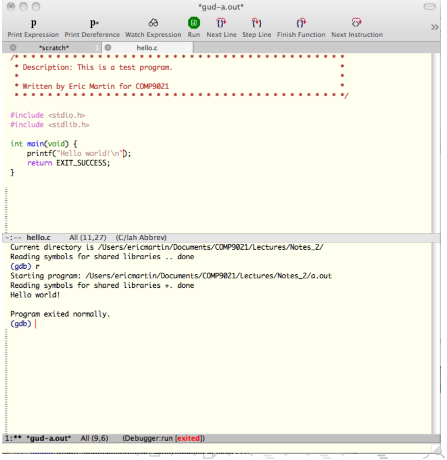


Using customised Emacs (6)

Reediting and retyping `\C-c p`



Typing `r` in bottom window



Programming in style (1)

I provide you with a perl script and a style sheet that will help you choose a particular programming style following a number of constraints, check that the programs you write are consistent with this style, and make a preliminary check that the logic of your program is not too complicated. This script will also be used to automark your assignments and partially assess their stylistic quality...

The file `mycstyle` is conveniently kept in the directory where the scripts previously discussed are kept. The file style_sheet.txt can be stored anywhere; a convenient location is `~/COMP9021`.

The script `mycstyle` expects to be given both the path to the file that contains the source code whose style you want to check, and the path to (possibly a copy of) the file `style_sheet.txt`.

Programming in style (2)

The following interaction from the command line shows that hello_world.c is stylistically correct.

```
$ mycstyle hello_world.c ~/COMP9021/style_sheet.txt
No stylistic mismatch has been detected.
```

If you would prefer to indent lines by 3 spaces rather than by 4 spaces, then you would change 4 to 3 on the 14th line of style_sheet.txt, namely

```
Number of spaces for indentation (may vary between 3 and 5)    3
```

In order to let Emacs automatically indent by 3 spaces rather than 4, you would remove one space before return EXIT_SUCCESS in _mctemplate and add the following to your .emacs.el file:

```
(add-hook 'c-special-indent-hook
          '(lambda ()
             (setq c-basic-offset 3)))
```

Programming in style (3)

The previous change to style_sheet.txt would make hello_world.c syntactically incorrect, with the details in hello_world.style.txt:

```
$ mycstyle hello_world.c ~/COMP9021/style_sheet.txt
2 stylistic mismatches have been detected.

The details are recorded in hello_world.style.txt
$ cat hello_world.style.txt
2 indentation mismatches have been detected.
This has to be fixed for indentation levels to be checked.


int main(void) {
@@    printf("Hello world!\n");
@@    return EXIT_SUCCESS;
}
```

A few useful Emacs commands (1)

- To exit Emacs, type `\C-x \C-c` Control x followed by Control c, which can be obtained by pressing the Control key, and then typing x followed by c (keeping the Control key pressed).
- If you make a mistake when you enter a command in the minibuffer window, type `\C-g` (Control g) once or twice.
- If you want to interrupt a running program, type `\C-c` (Control c) one or twice in the `gud-a.out` window.
- To select a file, create an associated buffer and display its contents in a window, type `\C-x \C-f` (Control x followed by Control f).
- To save the contents of a buffer to the file it is associated with, type `C-x \C-s` (Control x followed by Control s).
- To list all buffers that exist, type `\C-x \C-b` (Control x followed by Control b).
- To create or switch to a buffer, type `\C-x b` (Control x followed by b) and enter the name of the buffer as prompted in the minibuffer.

A few useful Emacs commands (2)

- To split a window and create a new window, type `\C-x 2`
- To have only one window open and close all others, type `\C-x 1` in the window you want to keep.
- To delete the character before the cursor, type `DEL` (press the delete key).
- To delete the character after the cursor, type `\C-d`.
- To delete the part of the word before the cursor, type `\M-DEL` (Meta Delete, with Meta being sometimes bound to the Option key, or to the Alt key).
- To delete the part of the word after the cursor, type `\M-d` (Meta D).
- To insert a comment, type `\M-;` (Meta semicolon).
- To get help and find out more about Emacs, type `\C-h ?`.

Easily compiling a program from the command line (1)

A program such as hello_world.c can be compiled from the command line by invoking gcc with the right options, the minimal command being:

```
gcc -std=gnu99 hello_world.c
```

Another option is to use the mmakefile perl script, that can be kept in the same directory as the other scripts, to automatically create a Makefile, and then type make to compile the program and create a.out, before running the program by typing a.out (possibly with command line arguments).

```
$ mmakefile hello_world.c
$ make
gcc -std=gnu99 -Wall -ggdb  hello_world.c
$ a.out
Hello world!
```

Easily compiling a program from the command line (2)

The `mmakefile` script does from the command line the same work as `_mmakefile` when automatically called from customised Emacs; both produce the same Makefile, that can be displayed from the command line:

```
$ cat Makefile
# Makefile produced by _mmakefile

CC = gcc
LDFLAGS =
CFLAGS = -std=gnu99 -Wall -ggdb

sources = hello_world.c

a.out : $(sources)
        $(CC) $(CFLAGS) $(LDFLAGS) $(sources)
```

Another option to compile a program from the command line is to write a Makefile in an editor, before calling `make`.

Prompting for input

Getting input from the user is very common; having users who do not always provide the expected input is very common. The source file p_io.c and associated header file p_io.h make the task of getting correct input from the user easier, making available 4 customised input functions

```
p_scanf()          p_fscanf()
p_prompt()         p_guarded_prompt()
```

which are precisely described in the comments at the top of `p_io.h`, but these descriptions will make full sense only later in the session. Of all those functions, `p_prompt()` is probably the most useful of all. Using it is easy, as illustrated by the program on the last slide. The function will prompt you relentlessly until you provide a valid input...

To indicate the end of input, the user is expected to press carriage return and then type `\C-d` (Control D).

Things to do to be able to use those functions

- In the `COMP9021` subdirectory of your home directory, create two directories:
  - `include`
  - `lib`
- In the directory where you saved `p_io.h`, execute the following

```
cp p_io.h ~/COMP9021/include/p_io.h
```

- In the directory where you saved `p_io.c`, execute the following

```
gcc -std=gnu99 -c p_io.c
gcc -shared -o ~/COMP9021/lib/libp_io.so p_io.o
rm p_io.o
```

- Edit the `.profile` file in your home directory, adding a line that reads `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/COMP9021/lib` (and run `. .profile` if you want the change to take effect without having to log out first).

Solving and implementing a solution to our first interesting problem

The program tower_and_2_glass_marbles.c allows us to tackle an interesting problem, to get a feel for what comes ahead, and to have a first look at many interesting aspects of the C language.

> *The user is prompted to enter the number of floors of a building. Using 2 marbles, one has to discover the highest floor, if any, such that dropping a marble from that floor makes it break, using a strategy that minimises the number of drops in the worst case (it is assumed that any marble would break when dropped from a floor when one marble breaks, and also when dropped from any higher floor; the marbles might not break when dropped from any floor).*