CP Lab 3: Programming, with some iteration

Instructions

The purpose of this Lab is to help you get more programming experience with C, and introduce you to using *iteration* or *loops* in your programming. We will be building on what you have done in the previous lab and in your classes.

You will use the `if`-statement construct and also the `for` and `while` loop constructs.

We know some of you did not all finish the `whatday.c` program within Lab2 last week. We hope you have made progress in developing and debugging that program in your own time. Please now take the opportunity to submit your 'What Day' if you haven't already done so, or, if you still have some small issues to resolve, ask for help with this. If you're very stuck with it, then leave it for the moment, and come back to it after this lab, or during the catch-up week in week 6.

All Labs for CP will take place on the DICE machines in AT5.05 (Thursday and Tuesday) or AT6.06 (Monday). For working alone, you may use any of the DICE labs in AT, as long as they are not booked out by a class. You have 24-hour access to AT levels 3,4,5: if your card/pin does not allow you in, use the ITO contact form on our course webpage to contact the ITO.

Aims

- Get you extra experience with the DICE system, and its use for C programming.
- Get you more familiar with the basics of imperative programming.
- Reinforce the idea of 'program planning'.
- Allow you to finish your `whatday.c` program from the sheet for Lab2, and submit it.
- Write a 'time-conversion' program making use of the basic concepts of assignment-to-variables, input, output and the `if`-statement (with a *complex* boolean condition).
- Give you extra experience of using certain parameters with `printf`, including showing you how to print a value in a fixed-width format.
- Use the return value from `scanf` to detect whether input succeeded.
- Give you experience of using the `for`-statement and the `while`-statement.

You will need to complete *everything* in this Lab sheet in order to establish a firm foundation for C programming.

Prerequisites

You should have attended your scheduled Lab session in week 3, and completed Lab sheet 2 during that session. *If not*, please catch up on Lab sheet 2 before proceeding with this one (unless you're stuck on `whatday.c`, in which case leave it till later).

Stage A Getting started

- Log into one of the DICE computers, and get a terminal in your home directory as in previous labs. From the home directory, create a subdirectory called `lab3` by entering the following from the command line:

  `mkdir lab3↵`

- We will not *immediately* move into the `lab3` directory. In Lab2, we discussed the development of a *template* program `template.c`. You should have created this file during Lab2. Assuming you have `template.c` created in your `lab2` sub-directory[1], you can make a copy of that to appear inside the `lab3` directory, as follows:

  `cp lab2/template.c lab3/↵`

  If you did *not* create `template.c` in Lab 2, please return to the sheet for Lab2 and work through 'Stage 2: Program Planning' on that sheet, before continuing with this lab.

- – If you still have work to do on `whatday.c` from Lab2, move into the `lab2` sub-directory to complete this [2]

    `cd lab2↵`

  – Otherwise, more likely, you are ready to start work on the new programming tasks for Lab3, in which case, move straight into the `lab3` sub-directory:

    `cd lab3↵`

- In doing your work ...
  * Develop your programs carefully
  * Please ask for help when you want it!
  * Try to tick off each of the tasks.

Stage B Clock

Time is often represented in 24-hour format; however, it can also be represented in 12-hour format, with the added notation of AM and PM. In this first program we will consider the problem of converting from 24-hour format to 12-hour format.

This task is in some sense similar to the 'What day' program at the end of Lab2; however, this is an easier task.

Your program must do the following steps (i)–(iv):

(i) Ask the user to enter a time in 24 hour format. This has three stages:
  - Use `printf` to tell the user your program will ask for a time in 24-hour format.
  - Use `printf` to ask for the hour (in 24-hour format) and read this with `scanf`.
  - Use `printf` to ask for the minutes and read this with `scanf`.

[1] If you have got your original `template.c` saved within a different directory than `lab2`, then you will have to use a different path-and-name to `lab2/template.c`, please ask your Demonstrator to help, if you have difficulty.

[2] Once you are done, you will later type `cd ~/lab3↵` to move to the `lab3` directory to continue with Lab3.

(ii) Next the 'error checking' step. Write an `if`-statement where the *boolean condition* tests whether the inputs represent a correct time in the 24-hour clock, to output an error message if it is not correct; and, otherwise to enter the 'main branch' of the `if`-statement, and compute the 12-hour equivalent.

  – The *boolean condition* for the `if` will need to be a complex boolean condition using `&&` or `||`, because we will need to check minutes as well as hours.

(iii) Within the 'main branch' of your program, you must compute whether or not the time is 'AM', 'PM', 'noon' (12h 00min) or 'midnight' (00h 00min). Then you must print out (using `printf`) the time in the form `**:** AM` or `**:** PM`, `12:00 noon` or `12:00 midnight`. (Technically, 00:00 is 12:00 AM and 12:00 is 12:00 PM, but many people find this confusing.)

  – Note that we will need *another* `if`-statement to detect these different cases and create the appropriate output for each case. So we will end up with a program using 'nested `if`-statements' (where one new `if`-statement is created inside a 'branch' of a previous `if`-statement).

(iv) Finally at the end we must 'return' `EXIT_SUCCESS`.

For this task we have provided a handwritten plan of the program structure, similar to that used in a couple of earlier lectures. This 'program plan' is included on the next page, and the sections where (i), (ii), (iii) and (iv) must go are labelled on the plan. The plan also has extra detail in the form of 'regions' for different sections of code, and some text explaining what will be done in that 'region'.

You may find it helpful to fill in this basic plan with the details of your program, *before* you go near the computer.

Your first step will be to plan which variables you will need (certainly two for the inputs of hours and minutes, but maybe others also), and define these in the 'variable declarations' region.

Then work through the different phases (i), (ii), (iii), (iv) of the program, adding your code to the plan in handwritten format. Look back on your lecture notes to make sure you get the 'syntax' (the way of writing statements in C) correct.

Finally, once you are happy with your code, turn to your computer to type-up, compile and run.

`printf` and `scanf` notes:

- Use of `scanf` to enter input: remember to use the 'magic ampersand' when scanning in – for example, to read an integer into the `int` variable `x`, we use

  ```
  scanf("%d", &x);
  ```

- You can use `printf` to output integers in fixed-width format, with leading zeros.

  For example, use `%03d` instead of `%d` if you want to print an integer as exactly 3 digits long, padded (if necessary) with leading zeros. The `%3d` tells it to use exactly three digits, the `0` in `%03d` means 'pad with leading zeros'. (If the integer is > 999, all its digits will be printed, even though that's more than 3 digits.)

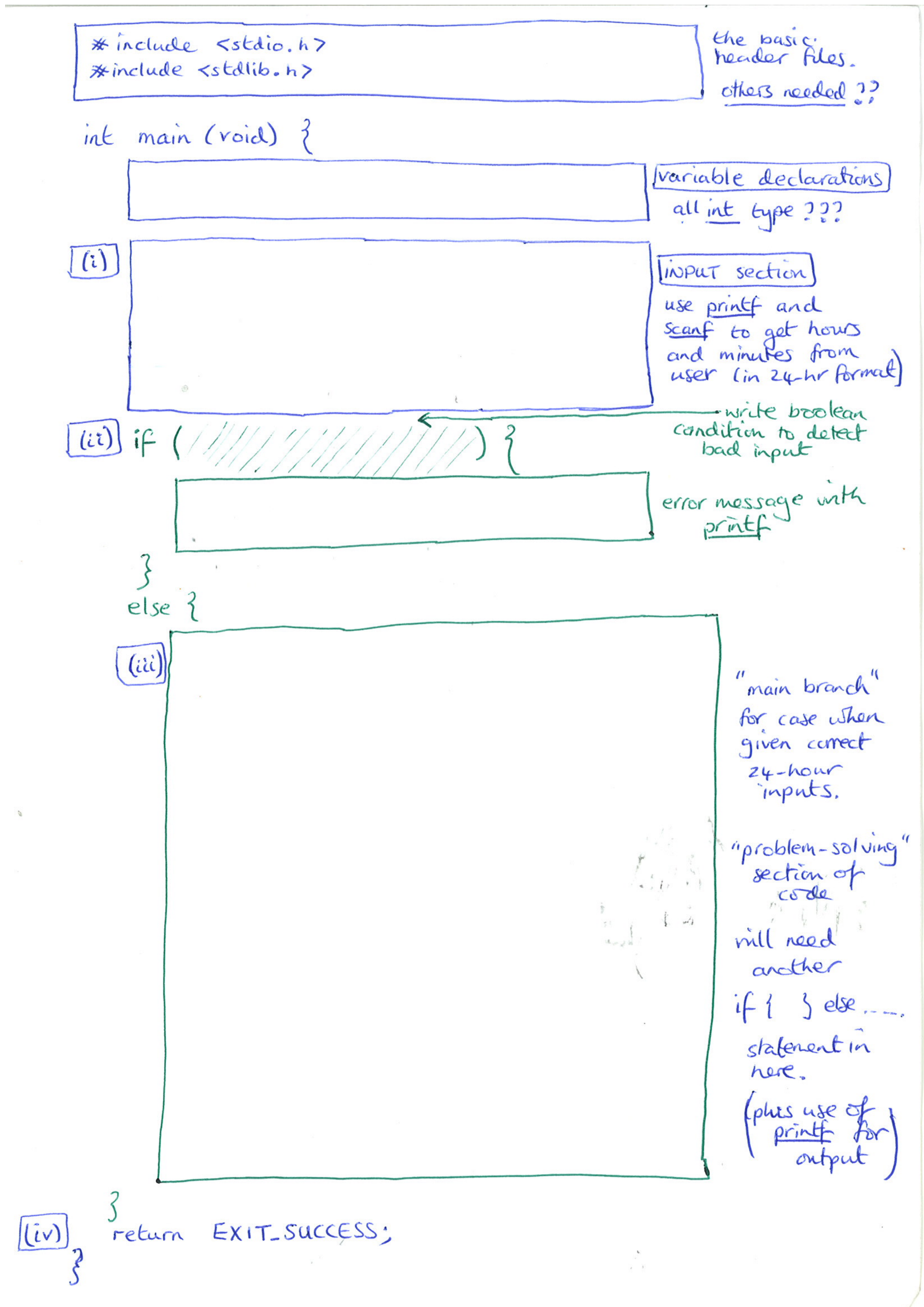

Figure 1: 'Program plan' for `timeformat.c` for you to complete.

Typing-up, compiling and running your code:

- Write your code in a program called `timeformat.c`.

  There is an easy way to start off this file, using the Unix copy command `cp` from the Terminal prompt as follows[3]:

  `cp template.c timeformat.c↲`

  Then subsequently open `timeformat.c` in emacs to develop your code further:

  `emacs timeformat.c &↲`

- Save your program when you are happy with it. Compile your program from the terminal prompt:

  `gcc -Wall timeformat.c↲`

  There may be some DEBUGGING to be done.

- When compilation is successful, the 'executable' will be created in the `a.out` file in your directory.

  Type `./a.out`, followed by ↲, to *run* your program and test its results. You may need to modify the program further.

- Move your executable `a.out` for `timeformat.c` into a different file, for example `timeformat`, for future use.[4] Then in the future, to 'run' the timeformat program, you will type the following:

  `./timeformat↲`

- Finally, submit your program with

  `submit cp lab3 timeformat.c↲`

Your result for `timeformat.c`

- ☐ A program which requests two input integers with `printf`, reads the inputs with `scanf`, and outputs with `printf`
- ☐ Program should be *correct*. To test:
  - Given the value 0 for hours and 20 for minutes, it should print 12:20 AM
  - Given the value 12 for hours and 35 for minutes it should print 12:35 PM
  - Given the value 13 for hours and 20 for minutes, it should print 01:20 PM
  - Given the value 18 for hours and 3 for minutes, it should print 06:03 PM.
- ☐ Apart from the problem-solving and use of `if`, you have also learned how to use `printf` to print out integers in a fixed-width format, and padded with leading 0s (see end of page 3 for this).
- ☐ Make sure you move `a.out` into `timeformat` before proceeding with the other tasks.
- ☐ Don't forget to submit your source file.

[3]Note we are assuming you made a copy of `template.c` within the `lab3` sub-directory, as part of the 'Getting started' section.

[4]Use the Unix 'move' command `mv` to do this.

Stage C Average of Two Numbers

Next write a simple program `averageof2.c` which takes as input two integers, and prints their average *to two decimal places*. As always a good start is to copy over `template.c` to `averageof2.c`

Recall in the previous lab you have learned how to work with real numbers in C (using `double`) and how to print these out via `printf`. In particular, recall that printing to two decimal places can be done with the `"%.2lf"` format for `printf`.

This program is a simple warm-up for stage D and E, when we will use the `for` and `while` constructs to solve a general version of 'average'.

Your result for `averageof2.c`

- ☐ Program should be *correct*. To test:
  - Given the values 5 and 15, it should return 10.00
  - Given the values 8 and 9, it should return 8.50
- ☐ Make sure you rename `a.out` into a file `averageof2` (or similar) before proceeding.
- ☐ Submit your file as usual with

  ```
  submit cp lab3 averageof2.c↲
  ```

Stage D Average of many numbers

Now we are going to write a program to solve a *general version* of the 'average' problem, where we will consider not just two inputs, but an *arbitrary number*[5] of integers to average. The value output must be the average of the numbers, to *two decimal places*.

To solve this problem it will be necessary to *use the concept of iteration* (either using `for` or `while`), which was introduced in lecture 6 of CP.

In this stage we will assume that the first stage of your program involves asking the user up-front 'How many integers' they want to average. We will see that this will allow you to use the `for`-statement to deal with the main part of the computation. Later in Stage E we will write a program which does not need the number of integers to be given at the beginning.

The program for this stage will be called `averagewithfor.c`.

The main steps of your program will be:

(i) First ask the user how many items (integers) they want to calculate the average of, and read this value.

(ii) Then write a `for`-loop, which will repeatedly ask for a new number, read this number, and add it to a 'running total'.

'Repeatedly' here means a number of repetitions which matches the value of the first integer given by the user in (i).

[5] In Maths/Computer Science, 'arbitrary' is a commonly-used word. It means 'any'.

(iii) On exiting the `for`-loop, the program must use the 'running total', and the first input from (i), to compute the average, and output this using `printf`.

Things to take into account for `averagewithfor.c`:

- It may be helpful for you to draw up a 'program plan' for `averagewithfor.c` (as we did on page 4 for `timeformat.c`), and to develop your code on paper first.
- Remember as always you can create a 'shell' for `averagewithfor.c` by copying over `template.c` in the usual way (`cp template.c averagewithfor.c`↵). Or you may prefer to start with a copy of `averageof2.c`.
- Your variables will be declared below the `main`. For this program, there are many 'variable issues':

  You will need a variable for the 'number of integers to average' which will be read from the user.
  You will need a variable to read in the different integers taken in for averaging.
  You will need a variable for the 'Running total'.
  You will need to *initialise* the 'Running total' variable to 0 at the beginning of your program. *Initialisation* is always an issue for variables which are going to be updated in any way different from a direct 'read-in' with `scanf`.
  You will need a 'loop counter' variable to use in your `for`.[6] The 'loop counter' variable is *usually* an `int`, and often is named `i`, `n` or `count` in examples.
  Finally, you will need a variable to store the average. Your average will not necessarily be an integer, so you need to consider which *type* you will use for this variable (namely `double`). Also you must use *casting* of `int` variables when you are using them in an expression to create a `double`.

The usual commands are used to do the compilation and execution of the program. To compile:

`gcc -Wall averagewithfor.c`↵

Once you have compiled your program (and the executable has been created in `a.out`) you can run it as usual with `./a.out`↵

Once you have debugged your program, save it for future execution in a file called `averagewithfor` (different from `averagewithfor.c`, no `.c` extension) using the Unix 'move' command `mv`.

When you're satisfied, submit your program in the same way as the others.

First 'Average of Many' program (`for`)

☐ Testing.

- You should get the result 8.00 in response to inputting 3 first, then the 3 numbers 7, 8 and 9
- You should get the answer 2.75 in response to inputting 4 first, then the 4 numbers 6, 4, −7 and 8

  Do further tests to check correctness.

☐ Remember to submit your program.

☐ If you have time, extend your program to test that the first integer read, the 'number of integers to average', is at least 1.

[6] Check back through the Lecture 6 slides, or look at section 4.9 of 'A book on C', to remind yourself what this means, and how we use it.

Use an `if`-statement to branch into either an error message (if the value is 0 or negative), or alternatively proceed with the main computation.

Stage E Average of many numbers II

In this next stage, we again consider the problem of averaging an arbitrary number of integers; however we will *not* ask for the number of integers at the start. Instead we will write a program containing a 'loop' where on each iteration, we try to read an integer. If we fail, we print the current average and finish; if we successfully read an integer, we add it to the running total and count, and loop round again. To do this, we will use a feature of `scanf`: it returns a value telling us how many variables it successfully read.

For this task you will use a *while*-statement to implement the iteration (looping) in your program. Your program must be named `averagewithwhile.c`. The main steps of the program will be:

(a) Use `printf` to output a message saying 'Welcome. You will be asked to input integers one at a time. To finish, type a letter or any non-digit.'

(b) Write a `while`-loop, which will repeatedly

- Use `printf` to tell the user 'Input the next integer:', and `scanf` to read in this value.
- If `scanf` fails, then set the variable `again` to zero; otherwise add the value read to a 'running total', and *also* add 1 to a 'count of inputs'.[7]

'Repeatedly' here means 'until the boolean condition in the while loop fails'. The *Boolean condition* of your `while` should simply check whether the `int` variable `again` (which should have been initialized to `1`) is non-zero.[8]

(c) On exiting the `while`-loop, the program must use the 'running total', and the 'count of inputs', to compute the average, and output this using `printf`.

To complete the program, you need to know how to find out whether `scanf` successfully read an integer. As we said, `scanf` returns a value, which is the number of items successfully read. So after doing

```
int myint;
int ret;

ret = scanf("%d",&myint);
```

the value of `ret`[9] will be `1` if the user typed a number, or `0` if the user typed a non-number.

Second 'Average of Many' program (`while`)

☐ Testing.

You should get the result 8.00 in response to inputting `7↲8↲9↲x↲`

You should get the answer 2.75 in response to inputting `6↲4↲-7↲8↲x↲`

☐ Submit your program as with the others.

[7] Need to count up how many inputs, because we need this at the end in getting the average.
[8] It's possible to write this program without needing an explicit `again` variable. You're free to do this if you see how!
[9] `ret`, short for 'return value', is a traditional name for variables used in this way.

☐ Would it be possible to solve the ‘Average of many’ problem with a `for`-statement in the case where we do NOT ask for the number of integers at the start? Think about this.

Julian Bradfield after *Prerona Mukherjee and Mary Cryan, 5 Oct 2016*